

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Luigi Domenico Cecchini Soares

MEMORY-SAFE ELIMINATION OF SIDE CHANNELS

Belo Horizonte
2022

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Luigi Domenico Cecchini Soares

**ELIMINAÇÃO AUTOMÁTICA DE CANAIS LATERAIS SEM ACESSOS
INCONSISTENTES À MEMÓRIA**

Belo Horizonte
2022

Luigi Domenico Cecchini Soares

MEMORY-SAFE ELIMINATION OF SIDE CHANNELS

Final version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
2022

Luigi Domenico Cecchini Soares

**ELIMINAÇÃO AUTOMÁTICA DE CANAIS LATERAIS SEM ACESSOS
INCONSISTENTES À MEMÓRIA**

Versão final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte
2022

© 2022, Luigi Domenico Cecchini Soares.
Todos os direitos reservados.

Soares, Luigi Domenico Cecchini.

S676m Memory-safe elimination of side channels [manuscrito] /
Luigi Domenico Cecchini Soares. – 2022.
91 f. il.

Orientador: Fernando Magno Quintão Pereira.

Dissertação (mestrado) – Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 87-91

1. Computação – Teses. 2. Fluxo de Informação – Teses. 3. Criptografia de dados (Computação) – Teses. 4. Criptografia – Canais laterais – Teses. 5. Segurança da informação – Teses. I. Pereira, Fernando Magno Quintão. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*46(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

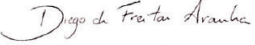
Memory-Safe Elimination of Side Channels

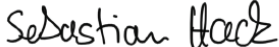
LUIGI DOMENICO CECCHINI SOARES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. MARIO SÉRGIO FERREIRA ALVIM JÚNIOR
Departamento de Ciência da Computação - UFMG


PROF. DIEGO DE FREITAS ARANHA
Department of Computer Science - Aarhus University


PROF. SEBASTIAN HACK
Department of Computer Science - Saarland University

Belo Horizonte, 3 de Março de 2022.

Dedico este trabalho aos meus pais, Marcelo e Silvana, que nunca mediram esforços para me apoiar. Sem eles, esta etapa da minha vida não seria possível.

Agradecimentos

Agradeço ao professor Luís Fabrício Wanderley Góes, meu orientador durante a graduação. Foi por meio dele que passei a conhecer o professor Fernando Magno Quintão Pereira, que hoje é meu orientador no mestrado. Ao professor Fernando, por todo o suporte e por todos os ensinamentos durante estes últimos dois anos. É, certamente, uma grande fonte de inspiração, como pesquisador, professor, orientador e como pessoa. Sem ele, nada do que foi desenvolvido neste projeto seria possível. Aos meus colegas do Laboratório de Compiladores (LaC), pelas experiências tanto profissionais quanto pessoais. Em especial, ao Michael Canesche, que teve participação direta e foi de suma importância para a realização dos experimentos apresentados neste trabalho. A todos os professores, tanto atuais quanto passados, que contribuíram para que pudesse chegar até aqui. Por fim, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pelo suporte financeiro dado ao longo destes dois anos.

"I'm not crazy. My mother had me tested."
(Sheldon Cooper)

Resumo

Um programa é dito isócrono se seu tempo de execução não depende de informações sensíveis. Isocronicidade é uma propriedade essencial em implementações criptográficas, posto que programas isócronos não apresentam vazamento de informações relacionadas a seus tempos de execução. Nesta dissertação, nós demonstramos como adaptar para o contexto de resistência à canais laterais um algoritmo de linearização parcial de grafos de fluxo de controle que foi, inicialmente, concebido para maximizar o desempenho em programas vetorizados. Esta transformação é correta: dada uma instância das entradas públicas, o programa parcialmente linearizado sempre executa a mesma sequência de instruções, independente das entradas secretas. Caso o programa original seja publicamente seguro, os acessos à cache de dados serão indistinguíveis no código transformado. Esta transformação é, também, ótima: todo desvio dependente de dados secretos é linearizado; nenhum desvio que dependa apenas de dados públicos é linearizado. Assim, a transformação preserva laços que dependem de informações públicas. Se todos os desvios que saem de um laço dependem de dados sensíveis, o programa modificado não terminará. Nossa transformação estende trabalhos recentes de maneiras não triviais. Ela é capaz de lidar com construções como “goto”, “break”, “switch” e “continue”, que não estão presentes na linguagem de domínio específico FaCT (2018). Assim como a ferramenta Constantine (2021), nossa transformação garante invariância de operações, mas sem necessitar de informações provenientes da execução dos programas. Além disso, em contraste com SC-Eliminator (2018), nossa técnica é capaz de lidar com programas contendo laços sem limites conhecidos em tempo de compilação.

Palavras-chave: Canal Lateral. Fluxo de Informação. Criptografia. Transformação de Programas.

Abstract

A program is said to be isochronous if its running time does not depend on classified information. Isochronicity is an essential property in cryptographic implementations, for isochronous programs do not leak time-related information. In this thesis, we demonstrate how to adapt to the context of side-channel resistance a partial control-flow linearization algorithm initially conceived to maximize work performed in vectorized programs. This transformation is sound: given an instance of the public inputs, the partially linearized program always runs the same sequence of instructions, regardless of the secret inputs. Incidentally, if the original program is publicly safe, accesses to the data cache will be data oblivious in the repaired code. This transformation is also optimal: every branch that depends on some secret data is linearized; no branch that depends on only public data is linearized. Thus, the transformation preserves loops that depend on public information. If every branch that leaves a loop depends on secret data, then the transformed program will not terminate. Our transformation extends recent work in non-trivial ways. It handles C constructs such as “goto”, “break”, “switch” and “continue”, which are absent in the FaCT domain-specific language (2018). Like Constantine (2021), our transformation ensures operation invariance, but without requiring profiling information. Additionally, in contrast to SC-Eliminator (2018), our implementation handles programs containing general, unbounded loops.

Keywords: Side Channel. Information Flow. Cryptography. Program Transformation.

Resumo Estendido

Um programa é dito isócrono se seu tempo de execução não depende de informações sensíveis. Isocronicidade é uma propriedade essencial em implementações criptográficas, posto que programas isócronos não apresentam vazamento de informações relacionadas a seus tempos de execução. Fontes de variação no tempo de execução de um programa, que causem vazamento de dados confidenciais, são conhecidas como canais laterais baseados em tempo. Existem muitos trabalhos na literatura no âmbito tanto da detecção quanto da remoção de tais canais. Ainda assim, a implementação de uma transformação de código estática, buscando a proteção de programas gerais contra ataques baseados em tempo de execução, permanece um problema em aberto.

Nesta dissertação, nós demonstramos como adaptar para o contexto de resistência à canais laterais um algoritmo de linearização parcial de grafos de fluxo de controle que foi, inicialmente, concebido para maximizar o desempenho em programas vetorizados. Esta transformação é correta: dada uma instância das entradas públicas, o programa parcialmente linearizado sempre executa a mesma sequência de instruções, independente das entradas secretas. Caso o programa original seja publicamente seguro, os acessos à cache de dados serão indistinguíveis no código transformado. Esta transformação é, também, ótima: todo desvio dependente de dados secretos é linearizado; nenhum desvio dependente apenas de dados públicos é linearizado. Assim, a transformação preserva laços que dependem de informações públicas. Se todos os desvios que saem de um laço dependem de dados sensíveis, o programa modificado não terminará.

Isocronicidade pode ser definida em termos de canais teóricos de informação, um conceito utilizado extensivamente nos campos da Teoria da Informação e — mais próximo ao contexto deste trabalho — Fluxo de Informação Quantitativo. Isto é, um programa determinístico é caracterizado como isócrono apenas quando, para qualquer instância das entradas públicas, o canal correspondente mapeia todas as instâncias de entradas secretas para o mesmo traço de operações e acessos de memória. Diz-se que um canal determinístico D refina outro canal C se, e somente se, D induz uma partição menos complexa (i.e. com menos grupos) que aquela induzida por C . No contexto de vazamento de informações, isto significa que o canal D nunca vazava mais do que o canal C . Para programas que são seguros com sombra — uma forma mais fraca de segurança pública — a abordagem descrita nesta dissertação produz código que refina sua contraparte original.

Nossa transformação estende trabalhos recentes de maneiras não triviais. Ela é capaz de lidar com construções como “goto”, “break”, “switch” e “continue”, que não estão

presentes na linguagem de domínio específico FaCT (2018). Assim como a ferramenta Constantine (2021), nossa transformação garante invariância de operações, mas sem necessitar de informações provenientes da execução dos programas. Além disso, em contraste com SC-Eliminator (2018), nossa técnica é capaz de lidar com programas contendo laços sem limites conhecidos em tempo de compilação.

Palavras-chave: Canal Lateral. Fluxo de Informação. Criptografia. Transformação de Programas.

Extended Abstract

A program is said to be isochronous if its running time does not depend on classified information. Isochronicity is an essential property in cryptographic implementations, for isochronous programs do not leak time-related information. Sources of time variance that cause sensitive data to leak are known as time-based side channels. There is much work in the literature related to both the detection and mitigation of side channels. And yet, the implementation of a static code transformation to protect general programs against timing attacks remains an open question.

In this thesis, we demonstrate how to adapt to the context of side-channel resistance a partial control-flow linearization algorithm initially conceived to maximize work performed in vectorized programs. This transformation is sound: given an instance of the public inputs, the partially linearized program always runs the same sequence of instructions, regardless of the secret inputs. Incidentally, if the original program is publicly safe, then accesses to the data cache will be data oblivious in the repaired code. This transformation is also optimal: every branch that depends on some secret data is linearized; no branch that depends on only public data is linearized. Thus, the transformation preserves loops that depend on public information. If every branch that leaves a loop depends on secret data, then the transformed program will not terminate.

Isochronicity can be defined in terms of information-theoretic channels, a concept extensively used in the fields of Information Theory and — closer to the context of this work — Quantitative Information Flow. That is, a deterministic program is characterized as isochronous just when, for any instance of the public inputs, the corresponding channel maps all instances of the secret inputs to the same trace of operations and memory accesses. A deterministic channel D is said to refine another channel C if, and only if, D induces a partition that is coarser than that induced by C . In the context of information leakage, this means that D never leaks more than C . For programs that are shadow safe — a weak version of public safety — the approach described in this thesis produces code that refines its original counterpart.

Our transformation extends recent work in non-trivial ways. It handles C constructs such as “goto”, “break”, “switch” and “continue”, which are absent in the FaCT domain-specific language (2018). Like Constantine (2021), our code transformation ensures operation invariance, but without requiring profiling information. Additionally, in contrast to SC-Eliminator (2018), our implementation handles programs containing general, unbounded loops.

Keywords: Side Channel. Information Flow. Cryptography. Program Transformation.

List of Figures

1.1	Functions <code>oFdF</code> , <code>oFdT</code> , <code>oTdF</code> and <code>oTdT</code> compare the user's guess <code>g</code> with a secret password <code>pw</code> . (a) <code>oFdF</code> returns immediately whenever two elements are different; as such, it is neither operation nor data invariant. (b) <code>oFdT</code> always performs the same number of comparisons; hence, it is data invariant, but it still is not operation invariant. (c) <code>oTdF</code> is operation invariant; however, it has indirect accesses through a table <code>t</code> , using secret-dependent indices (<code>pw[i]</code>), and thus it is not data invariant. (d) <code>oTdT</code> always performs the same sequence of instructions and memory accesses; thus, it is both operation and data invariant.	24
3.1	Dominance relation: every path from s — the unique start node — to node v must go through node u ; that is, u dominates v	35
3.2	Post-dominance relation: every path from node u to d — the unique exit node — must go through node v ; that is, v post-dominates u	36
3.3	The influence region of node u is composed by all node in paths from u to its post dominator v ; that is, all nodes in the colored region.	36
3.4	(a) Partition induced by $A^{n=1, g=[0]}$, which corresponds to function <code>oFdF</code> from Figure 1.1 (a). (b) Partition induced by $D^{n=1, g=[0]}$, which corresponds to function <code>oTdT</code> from Figure 1.1 (b). Channel $A^{n=1, g=[0]}$ is refined by $D^{n=1, g=[0]}$, i.e. $D^{n=1, g=[0]}$ leaks no more than $A^{n=1, g=[0]}$	39
4.1	Syntax of the baseline language used throughout the thesis to design the isochronous transformation.	42
4.2	Control-flow graph of the password comparison function <code>oFdF</code> seen in Figure 1.1 (a), written in our baseline language.	43
4.3	Recursive definitions of block and edge conditions. Function <code>predecessors(ℓ)</code> gives all the blocks that are predecessors of block ℓ . Function <code>terminator(ℓ)</code> returns the last instruction of the block labeled by ℓ (see Figure 4.1).	43
4.4	CFG of function <code>oFdF</code> from Figure 4.2, with $n = 2$ and the loop unrolled. Edges are labeled with edge conditions (Definition 4.1, Figure 4.3).	44
4.5	Transformation rule for linearizing the CFG of a program. Function <code>rewrite_{br}</code> replaces every branch with another branch whose target is the next node in the topological order. Function <code>topological(G)</code> returns the topological sort (Definition 3.7) of the acyclic graph G	45

4.6	Linearization of the CFG from Figure 4.4, following the rewrite rule defined in Figure 4.5. The topological sort adopted for this transformation was: <code>begin, if.0, if.1, ret.false, end.</code>	45
4.7	Transformation rules used to rewrite phi functions, pre-PCFL. Function $rewrite_\phi$ takes a phi node and returns a combination of <code>ctsets</code> that are equivalent to that phi node. If the phi node has only one incoming value, it is equivalent to a simple assignment, and thus do not need any modifications. . .	46
4.8	(a) CFG before linearization. (b) CFG after linearization, with the phi function at block <code>g</code> rewritten according to Figure 4.7; variables <code>ec.d_g</code> and <code>ec.e_g</code> store the edge condition of, respectively, edges $d \rightarrow g$ and $e \rightarrow g$ from the original graph (see Definition 4.1, Figure 4.3).	46
4.9	Transformation rules for memory operations. Function $rewrite_{ld}$ makes loads memory safe, while function $rewrite_{st}$ makes stores both memory safe and sound with respect to whether they should or not take effect. Both rules rely on block conditions (see Definition 4.1).	47
4.10	(a) Original load from Figure 4.4. (b) Transformed load. (c) Example of a store. (d) Transformed store.	48
4.11	CFG from Figure 4.4 after linearization and with the instructions rewritten. (a) The shadow memory and the size of inputs <code>g</code> and <code>pw</code> , used in the transformation of loads. (b) Computation of block conditions (§4.2, Figure 4.3). (c) Predication of load instructions to ensure memory safety (§4.3.3, Figure 4.9). (d) Transformation of a phi function (§4.3.2, Figure 4.7).	49
4.12	Interprocedural transformations.	51
5.1	(a) CUDA kernel that counts occurrences of keys in a matrix. (b) Control-flow graph of the kernel. (c) Partially linearized control-flow graph.	53
5.2	Partial Control-Flow Linearization. <code>linearize(G)</code> produces a graph <code>GL</code> that is a partially linearized version of <code>G</code> . A preprocessing step removes the back edges in <code>G</code> before linearization; hence, <code>linearize</code> receives an acyclic graph. . .	55
5.3	Sequence of steps that function <code>linearize</code> in Figure 5.2 performs on the program from Figure 5.1, given that <code>Index = [0, 1, 3, 2, 4, 5]</code>	56
5.4	Recursive definitions of block and edge conditions for headers and tainted exiting edges. Function $predecessors_f$ is related to forward edges and $predecessors_b$ to back edges. When written without subscripts, $BC(\ell)$ (similarly for EC) refers to the last execution of block ℓ	59
5.5	(a) CFG from Figure 4.2; dashed arrows represent back edges; gray nodes are tainted. (b) The CFG with collapsed loop; numbers indicate the compact ordering. (c) Compact ordering of loop. (d) The collapsed CFG after linearization. (e) Whole CFG after linearization.	62

5.6	(a) CFG before partial linearization; block <code>b</code> is tainted. (b) CFG after partial linearization, with the phi function at block <code>g</code> rewritten; variable <code>ec.d_g</code> stores the edge condition of edge $d \rightarrow g$ from the original graph (see Definition 4.1, Figures 4.3 and 5.4).	65
5.7	Transformation rule for phi nodes. <i>inst @ ℓ</i> indicates that the instruction belongs to the block labeled by ℓ . $E(G)$ are the edges of graph G . <i>fold</i> relies on edge conditions (see Definition 4.1, Figures 4.3 and 5.4).	66
5.8	(a) Original load from Figure 4.2. (b) Transformed load. (c) Example of a store. (d) Transformed store.	67
5.9	CFG from Figure 4.2 after PCFL and with the instructions rewritten. (a) The shadow memory and the size of inputs <code>g</code> and <code>pw</code> , used in the transformation of tainted loads. (b) Computation of the block condition of the loop header (§5.3, Figure 5.4). (c) Computation of the edge condition of the tainted exiting edge <code>body → ret.false</code> (§5.3, Figure 5.4). (d) Predication of load instructions to ensure memory safety (§5.4.3, Figure 4.9). (e) Transformation of a phi function (§5.4.2, Figure 5.7).	70
6.1	Code size (in number of LLVM instructions) of transformed programs. Numbers on top show the size of original programs (compiled with LLVM 13.0 at the -O3 optimization level). Symbols in gray boxes show tools that are missing for particular benchmarks. <code>CTT</code> refers to <code>Constantine</code> , <code>SC</code> refers to <code>SC-Eliminator</code> . <code>Orig</code> refers to these two tools as originally implemented. <code>CFL</code> refers to these two tools with control-flow linearization only — thus, closer to our implementation of PCFL in purpose.	78
6.2	Time (in milliseconds) to apply each transformation onto the benchmarks. To give the reader some perspective on this comparison, the numbers on top show the time taken to run <code>LLVM opt -O3</code> on each benchmark. The gray boxes mark benchmarks that some tools could not handle.	79
6.3	Running time (in microseconds) of transformed programs. Numbers on top show time of original programs (compiled with LLVM 13.0). Symbols in gray boxes show missing tools for particular benchmarks.	81
6.4	Security guarantees achieved by the different tools. <code>Cor</code> indicates if the transformed program produces the same output as its original counterpart (i.e. if the transformed program is <i>correct</i>). <code>Data</code> refers to data invariance. <code>Opr</code> refers to operation invariance without compiler optimizations. <code>Opr3</code> refers to operation invariance at the <code>LLVM opt -O3</code> optimization level.	82

6.5	Comparison between programs written in C and linearized with PCFL, and similar programs written in FaCT, using equivalent control-flow structures. The column <code>.o</code> shows the size, in bytes, of the binary object file. The column <code>Instrs</code> shows the number of instructions in the LLVM representation of each program. Because they use different <code>main</code> functions, we show results with and without this routine.	84
-----	---	----

List of Tables

1.1	Summary of results from §6 with regard to the nine benchmarks that all the tools can handle. Numbers are arithmetic means. Measurements happen after programs are transformed and then optimized with LLVM <code>opt -O3</code> . <code>Original</code> refers to the benchmark without any transformation. <code>PCFL</code> and <code>Lif</code> correspond to our implementations. <code>CTT</code> refers to <code>Constantine</code> ; <code>SC</code> refers to <code>SC-Eliminator</code> . These two tools can do control- and data-flow linearization. Hence, <code>-Orig</code> refers to their original implementations, and <code>-CFL</code> refers to the implementation with only control-flow linearization. Our approaches only do control-flow linearization, but achieves data invariance for publicly-safe programs.	27
-----	--	----

Contents

1	Introduction	21
1.1	The Breakthroughs of 2021	21
1.2	Enter Partial Control-Flow Linearization	22
1.3	The Contributions of this Work	23
1.3.1	Threat Model	25
1.3.2	Comparison with Previous Work	26
1.4	Summary of Results	26
1.5	Publications	27
2	Literature Review	28
2.1	Partial Control-Flow Linearization	28
2.2	Side Channels	29
2.2.1	Detection of Side Channels	29
2.2.2	Mitigation of Side Channels	30
2.2.3	Constant-Time Preservation	32
3	Preliminaries	34
3.1	Graph Definitions	34
3.2	Quantitative Information Flow	36
4	Transforming Loop-Free Programs	41
4.1	Baseline Language	41
4.2	Predication	42
4.3	Rewriting System	44
4.3.1	Control Flow	44
4.3.2	Phi Functions	45
4.3.3	Memory Operations	46
4.3.4	Final Example	48
4.4	Interprocedural Transformation	50
5	From PCFL to SCE	52
5.1	Partial Control-Flow Linearization	52
5.1.1	Properties of PCFL	56
5.2	Taint Analysis	57

5.3	Predication	58
5.3.1	Accounting for Time	58
5.3.2	Active Paths	60
5.4	Rewriting System	60
5.4.1	Control Flow	61
5.4.2	Phi Functions	64
5.4.3	Memory Operations	66
5.4.4	Final Example	69
5.5	Correctness	71
5.5.1	Isochronification Preserves Semantics	71
5.5.2	Isochronification Implements Refinement	74
6	Evaluation	76
6.1	RQ1: Size of Transformed Code	78
6.2	RQ2: Transformation Time	79
6.3	RQ3: Performance of Transformed Code	80
6.4	RQ4: Security Evaluation	81
6.5	RQ5: Comparison with a Domain-Specific Language	83
7	Conclusion	85
	Bibliography	87

Chapter 1

Introduction

A program is said to be *isochronous* if, for any fixed instance of its *public* inputs, its running time remains the same regardless of its *secret* (sensitive) inputs. *Isochronicity* is characterized by two properties: data and operation invariance (see §1.3). Isochronous programs do not leak time-related information [Kocher, 1996]; therefore, isochronicity is an essential property in implementations of cryptographic routines [Almeida et al., 2016; Almeida et al., 2020; Barthe et al., 2019]. In view of this importance, much work has been done to detect time-variant code [Reparaz et al., 2017; Almeida et al., 2016; Ngo et al., 2017; Barthe et al., 2019; Guarnieri et al., 2020] or to remove sources of time variance [Agat, 2000; Almeida et al., 2020; Fell et al., 2019; Borrello et al., 2021; Cleemput et al., 2012; Van Cleemput et al., 2020; Gruss et al., 2017; Tizpaz-Niari et al., 2019; Chattopadhyay and Roychoudhury, 2018; Wu et al., 2018]. And yet, the implementation of a static code transformation technique capable of removing time-based side channels from programs containing general loops remains an elusive endeavor.

1.1 The Breakthroughs of 2021

Current methodologies to remove time-based side channels from a program consist in *linearizing* its control-flow graph. Linearization removes branches from a program. Until recently, the state-of-the-art approach to perform linearization was due to Wu et al. [2018] (**SC-Eliminator**). In 2021, we proposed **Lif** as an improvement of Wu et al.’s transformation, to prevent it from introducing out-of-bounds accesses into the program [Soares and Pereira, 2021]. Both our and Wu et al.’s technique are fully static: they do not require executing a program to change it. However, they cannot deal with programs containing loops, unless these loops have bounds known at compilation time.

Still in 2021, Borrello et al. [2021] introduced **Constantine** as a dynamic alternative to **Lif**’s static approach. Borrello et al. execute the program and use runtime information like memory addresses and the outcome of branches to linearize the part of

the code that could be covered during the execution. Borrello et al.’s strategy handles programs with general loops. To the best of our knowledge, it does not insert invalid memory accesses into programs. However, it also has limitations, the most serious being due to Rice’s Theorem [Rice, 1953]: it is undecidable to find inputs to exercise specific parts of a program’s code. Indeed, our personal experience with `Constantine` is that it is hard to find inputs that reach particular branches that should be linearized. In this thesis, we show that it is possible to handle programs with general loops with a static approach, hence joining the benefits from `Lif` and `Constantine`’s transformations.

1.2 Enter Partial Control-Flow Linearization

In 2018, Moll and Hack [2018] introduced partial control-flow linearization (PCFL): a code-optimization technique to speed up programs in the *Single-Instruction, Multiple-Data* (SIMD) model [Flynn, 1972]. A SIMD program is processed by multiple threads running in lockstep. The hardware fetches one instruction at a time, which is forwarded to all the threads. Thus, these threads process the same instruction simultaneously, albeit on different data. In a SIMD program, some branches can be proven to be *uniform*, meaning that they always yield the same outcome for the threads that execute them together. The other branches are called *divergent*.

Moll and Hack’s PCFL removes the divergent branches from the program, linearizing the blocks controlled by said branches. The transformation keeps the uniform branches unchanged. In principle, PCFL bears as much importance to side-channel resistance as the fact that more kangaroos live in Australia than people in Uruguay.¹ However, replace “uniform” with *public* and “divergent” with *secret*, and *voilà*: we have a beautiful algorithm to produce isochronous programs! To make this thesis self-contained, we present Moll and Hack’s PCFL algorithm in §5.1. Then, in §5.4.1 we show how we adopted PCFL in the context of software security.

¹<https://twitter.com/redditcfb/status/1355288917558390786>

1.3 The Contributions of this Work

One effective countermeasure to prevent time-based side channels is to write programs that do not perform secret-dependent branches and memory accesses, thus ensuring time invariance. Such paradigm is known as *Constant-Time Programming* and is adopted by several popular cryptographic libraries. Nevertheless, it requires much attention and knowledge to write constant-time programs by hand, since minor mistakes could be the source of vulnerabilities. In other words, such a manual job is error prone and therefore an automated method is preferable.

This thesis shows how to adapt partial control-flow linearization to make programs isochronous, thus ensuring *Cryptographic Constant-Time* (CCT) behavior [Barthe et al., 2021, §2.3]. To use our code transformation, users must indicate which program inputs are secret. No more interventions are necessary. The generated code achieves the following properties, which guarantee standard notions [Rafnsson et al., 2017, §4] of *confidentiality* and *non-interference*:

Operation Invariance: Given an arbitrary instance of the public inputs, every possible execution of the transformed program processes the same sequence of addresses in the instruction cache, independent of the secret inputs.

Data Invariance: Given an arbitrary instance of the public inputs, every possible execution of the transformed program processes the same sequence of reads and writes in the data cache, independent of the secret inputs — this holds whenever the original program is *publicly safe* [Cauligi et al., 2019, §3.2.3].

Memory Safety: The transformed program only contains out-of-bounds memory accesses that already exist in the original program.

Termination: A loop in the transformed program only terminates due to public information. A loop controlled only by secret data will not terminate.

Example 1.1 illustrates the concepts of operation and data invariance, while Example 1.2 explains the issue of memory safety.

Example 1.1. Functions `oFdF`, `oFdT`, `oTdF` and `oTdT` (`o` = operation, `d` = data, `F` = false, `T` = true), in Figure 1.1, compare an input `g` with a secret password `pw`.² Even though the three functions compute the very same thing, they behave quite differently. Function `oFdF`, from Figure 1.1 (a), returns immediately whenever `g[i] != pw[i]`. Thus,

²The code depicted in Figure 1.1 is merely used as an example throughout this thesis. Bear in mind that passwords should never be stored as plain texts.

<pre> 1 // g (guess) is public. 2 // pw (password) is secret. 3 // n is public. 4 int oFdF(int *g, int *pw, int n) { 5 for (int i = 0; i < n; i++) 6 if (g[i] != pw[i]) return 0; 7 return 1; 8 }</pre>	<p>a</p> <p>$\bar{O} \wedge \bar{D}$</p>
<pre> 1 // g (guess) is public. 2 // pw (password) is secret. 3 // n is public. 4 int oFdT(int *g, int *pw, int n) { 5 int r = 1; 6 for (int i = 0; i < n; i++) 7 if (g[i] != pw[i]) r = 0; 8 return r; 9 }</pre>	<p>b</p> <p>$\bar{O} \wedge D$</p>
<pre> 1 // g (guess) is public. 2 // pw (password) is secret. 3 // n is public. 4 int oTdF(int *g, int *pw, int *t, int n) { 5 int r = 0; 6 for (int i = 0; i < n; i++) { 7 // secret-dependent index: pw[i] 8 r = t[g[i]] != t[pw[i]]; 9 } 10 return r ? 0 : 1; 11 }</pre>	<p>c</p> <p>$O \wedge \bar{D}$</p>
<pre> 1 // g (guess) is public. 2 // pw (password) is secret. 3 // n is public. 4 int oTdT(int *g, int *pw, int n) { 5 int r = 0; 6 for (int i = 0; i < n; i++) 7 r = g[i] != pw[i]; 8 return r ? 0 : 1; 9 }</pre>	<p>d</p> <p>$O \wedge D$</p>

Figure 1.1. Functions `oFdF`, `oFdT`, `oTdF` and `oTdT` compare the user's guess `g` with a secret password `pw`. (a) `oFdF` returns immediately whenever two elements are different; as such, it is neither operation nor data invariant. (b) `oFdT` always performs the same number of comparisons; hence, it is data invariant, but it still is not operation invariant. (c) `oTdF` is operation invariant; however, it has indirect accesses through a table `t`, using secret-dependent indices (`pw[i]`), and thus it is not data invariant. (d) `oTdT` always performs the same sequence of instructions and memory accesses; thus, it is both operation and data invariant.

`oFdF` is neither operation nor data invariant. Procedure `oFdT`, on the other hand, is data invariant. The loop seen at lines 6–7 of Figure 1.1 (b) always executes the same number of iterations. Hence, the sequence of memory accesses is always the same, regardless of `pw`. Nevertheless, `oFdT` is still not operation invariant, since the execution of the assignment at line 7 depends on content of the array `pw`. Function `oTdF`, from Figure 1.1 (c), always performs the same sequence of instructions and the same number of memory accesses; however, some of these accesses rely on secret-dependent indices (`pw[i]`), implying that `oFdT` is not data invariant. Finally, function `oTdT`, from Figure 1.1 (d), always executes the same sequence of instructions and perform the same sequence of memory accesses, independent of the secret input `pw`. Therefore, `oTdT` is both operation and data invariant.

Example 1.2. The program repair proposed by Wu et al. [2018] would transform function `oFdF` (Figure 1.1 (a)) into code that is similar to function `oTdT` (Figure 1.1 (c)). The latter is both data and operation invariant; hence, it is isochronous. However, Wu et al.’s transformation is not memory safe, i.e. the repaired version might access unallocated memory even if such bad accesses do not occur in the original code with the same inputs. For instance, calling `oTdT` with `g = [0]`, `pw = [1]` and `n > 1` would incur in invalid accesses to `a[i]` and `b[i]`, $i > 0$, a fault that would never happen in function `oFdF` due to the early return at line 6.

1.3.1 Threat Model

We adopt a *string-of-addresses* threat model assumed by previous work [Cauligi et al., 2019; Soares and Pereira, 2021]. We assume an attacker who can observe the sequence of addresses accessed by a program in the instruction and data caches. In other words, the attacker has access to the trace formed by the memory addresses read or written by a program, including the address of the instructions fetched during execution. This model delivers stronger guarantees than the model typically adopted in cache-based timing attacks. The *hit-miss* model considers an attacker who has access to the sequence of cache hits and misses [Borrello et al., 2021; Wu et al., 2018; Zhang et al., 2022], assuming a *deterministic timing model* [Balliu et al., 2014, §3].

Both models, string-of-addresses and hit-miss, follow the general execution-trace model discussed by Zdancewic and Myers [2001, §2]; however, the string-of-addresses model includes less programs. As an example, any program that contains a memory access indexed by secret information, e.g. `a[secret[i]]`, will yield a different string of accesses in the data cache, thus leaking information even if only cache hits are verified in practice. In the words of Cauligi et al. [2019], these programs are not *publicly safe*. In

fact, an indirect comparison between these two models is available in the work of Zhang et al. [2022]. Zhang et al. compares two tools, `CAPE` and `Lif`. The former uses the hit-miss model; the latter, the string-of-addresses model. Out of five benchmarks transformed by `CAPE`, `Lif` could secure only one.

1.3.2 Comparison with Previous Work

As we shall see in §5, programs produced by our transformation still might have branches, as long as these branches are not controlled by sensitive information. Furthermore, our technique is capable of handling general loops. Therefore, we expand previous work in multiple ways:

1. *Static Generality*: In contrast to previous work [Soares and Pereira, 2021; Wu et al., 2018], our transformation handles programs with loops, even if these loops cannot be fully unrolled (i.e. are unbounded).
2. *Static Efficiency*: In contrast to previous static transformations [Soares and Pereira, 2021; Wu et al., 2018], we preserve branches controlled by public information, avoiding the unnecessary execution of unreachable code.
3. *Decidability*: Our transformation is fully static; hence, in contrast to a dynamic tool like `Constantine` [Borrello et al., 2021], it does not require test cases that exercise all the branches of a program.
4. *Convenience*: In contrast to a domain-specific language such as `FACT` [Cauligi et al., 2019], programmers can write memory-safe code directly in general-programming languages like C and still obtain isochronicity.

1.4 Summary of Results

We implemented our ideas in LLVM 13.0 [Lattner and Adve, 2004]. Chapter 6 compares these implementations with `Constantine`, `SC-Eliminator` and `FACT` in regard to 13 programs whose inputs can be split into public and secret data. Section 6.4 certifies that the transformed programs meet the guarantees previously enumerated, i.e. operation

invariance in general, data invariance for publicly-safe programs, memory safety and termination. Table 1.1 summarizes results reported in §6 for the nine benchmarks that all the tools can handle. `Lif` refers to the loop-free approach described in §4, whereas `PCFL` corresponds to our implementation of Moll and Hack’s partial control-flow linearization, presented in §5. `Lif` and `SC-Eliminator` cannot handle three benchmarks due to unbounded loops. These two tools and `Constantine` failed to produce correct output for another one. Notice that code size for `Lif` and `SC-Eliminator` is much bigger because these tools require that loops are fully unrolled.

Table 1.1. Summary of results from §6 with regard to the nine benchmarks that all the tools can handle. Numbers are arithmetic means. Measurements happen after programs are transformed and then optimized with LLVM `opt -O3`. `Original` refers to the benchmark without any transformation. `PCFL` and `Lif` correspond to our implementations. `CTT` refers to `Constantine`; `SC` refers to `SC-Eliminator`. These two tools can do control- and data-flow linearization. Hence, `-Orig` refers to their original implementations, and `-CFL` refers to the implementation with only control-flow linearization. Our approaches only do control-flow linearization, but achieves data invariance for publicly-safe programs.

Tool	Original	PCFL	Lif	CTT-Orig	CTT-CFL	SC-Orig	SC-CFL
Size (#LLVM instrs.)	330.78	337.00	15,342.11	464.67	365.89	10,863.56	8,444.89
Running time (μ s)	3.23	5.21	12.09	14.94	6.27	5.19	4.60
Linearization time (ms)		33.49	271.61	2,045.22	65.19	2,697.06	2,149.28

1.5 Publications

As mentioned in §1.1, we published, in 2021, at the International Symposium on Code Generation and Optimization (CGO 2021), an initial attempt to make programs isochronous statically [Soares and Pereira, 2021]. Such transformation is the topic of §4. In §5, we extend this transformation in non-trivial ways: we adapt Moll and Hack [2018]’s partial control-flow linearization to the context of side-channel elimination, and we augment the static analysis used for predication, initially described in §4.2, to handle general loops. These extensions constitute a manuscript that is currently under review [Soares et al., 2022]. Programs produced by the technique developed in §5 may still have branches, as long as these branches are not controlled by sensitive data; the same is not true for the more restrict, loop-free approach presented in §4.

Chapter 2

Literature Review

This work draws its contributions from two different communities: high-performance computing and software security. Concerning the former, this work is related to research about control-flow linearization. Concerning the latter, it is related to the static elimination of side channels. In this chapter, we explain how this thesis relates to previous contributions in these two domains. We begin by discussing, in §2.1, about partial control-flow linearization. Then, in §2.2, we move to the topic of side channels.

2.1 Partial Control-Flow Linearization

In its essence, Moll and Hack [2018]’s algorithm for control-flow linearization is an efficient way to support predication, inasmuch as it spares uniform branches from being predicated. Predication is, essentially, a technique to convert control dependencies into data dependencies. To the best of our knowledge, the first description of a systematic way to perform predication is due to Allen et al. [1983], although the problem had already been described in earlier work [Towle, 1976; Wolfe, 1978]. After Allen et al.’s original foray in the field, predication has been refined and expanded in many different ways, and today is standard textbook material [Clements, 2013]. In §4.2, we define a static analysis to compute the conditions that dictate which operations of a program shall be executed, which we then use to predicate instructions. Later on, in §5.3, we augment such a static analysis to handle general (unbounded) loops.

The fact that control-flow linearization was already a concern almost 40 years ago makes it surprising that Moll and Hack’s algorithm took so long to emerge. Compared to previous work, PCFL enjoys a number of advantages. First, when compared to Ferrante and Mace [1985]’s well-known linearization approach, Moll and Hack’s algorithm has better complexity (linear vs log-linear). Second, it is substantially simpler than previous approaches of similar service, such as Karrenberg and Hack [2012]’s. In the words of Moll and Hack: “*Karrenberg and Hack’s method spans over five algorithm listings*”, whereas the

PCFL routine is fully described by the 34 lines of Python in Figure 5.2. Finally, PCFL handles unstructured control flows, in contrast to heuristics used in practice [Moreira et al., 2017] by the Intel SPMD Compiler, for instance.

Nevertheless, we emphasize that this project is not about the design of a partial control-flow linearization approach. We reuse Moll and Hack’s algorithm almost without modifications. Our changes in the algorithm will be described in §5.4.1. There exists only one important difference between Moll and Hack’s implementation and ours, which is a consequence of the different purposes that we have when using PCFL. In Moll and Hack’s context, loops only terminate when all threads exit it; thus, the linearized loop contains only one exit block, at its end. Moll and Hack add phi functions to identify through which exit each thread left the loop. This approach is similar to what `Constantine` does: it computes an upper bound for the loop and forces execution up to this trip count. In our case, a loop can have multiple exits: indeed, any exit that is only controlled by public data will be left untouched by our transformation.

2.2 Side Channels

Timing attacks became the focus of much research during the nineties [Dhem et al., 1998; Kocher, 1996; Kocher et al., 1999; Wray, 1992]. However, the problem has been known before [Singel, 1976]. The literature contains many examples of both detection and defense mechanisms against such attacks. We thus shall split the discussion between these two fields of research: we first present, in §2.2.1, a few projects about the detection of side channels, and then, in §2.2.2, we explain how this work relates to previous side-channel mitigation techniques. We close this section by talking about the preservation of the constant-time property, which standard compiler optimizations might break — this relates to the concept of refinement from the Quantitative Information Flow theory [Alvim et al., 2020]; for more details, see §3.2.

2.2.1 Detection of Side Channels

Most of the literature concerning side channels refer to their identification, not to their elimination. For instance, Rodrigues et al. [2016] rely on the properties of the Static Single Assignment form to design an efficient detection method that operates on

the LLVM intermediate representation. Rodrigues et al. improve on the concept of control dependencies from Ferrante et al. [1987] by proposing a different approach that avoids Ferrante et al.’s worst case: $O(|I| \times |E|)$ dependency edges, where I are the instructions in a program and E are the edges in its control-flow graph. In addition, Rodrigues et al. provide an implementation of Hunt and Sands [2006]’s flow-sensitive system of security types, using their proposed notion of control dependencies. Rodrigues et al.’s work also has a practical contribution: `FlowTracker` is an LLVM-based tool that uses their implementation of Hunt and Sands’s type system to detect time-based side channels. As we shall see in §5.2, we use Rodrigues et al.’s information analysis to label variables as either tainted or non-tainted.

Around the same time (2016–17), Almeida et al. [2016] and Reparaz et al. [2017] introduced techniques to determine whether a code runs in constant time or not. Ngo et al. [2017] described a type system for verifying that a code correctly implements constant-resource behavior. More recently, Guarnieri et al. [2020] introduced a framework for specifying hardware-software contracts that assert which program executions an adversary can distinguish. A CPU satisfies a contract if, whenever two program executions agree on all observations, they are guaranteed to be indistinguishable by the adversary at the microarchitectural level.

2.2.2 Mitigation of Side Channels

There exists a wide range of approaches to mitigate information leakage due to time-based side channels [Ngo et al., 2017; Agat, 2000; Tizpaz-Niari et al., 2019; Wu et al., 2018; Rane et al., 2015; Cauligi et al., 2019; Soares and Pereira, 2021; Borrello et al., 2021]. The seminal work in the field is due to Johan Agat, who has proposed a type-directed transformation to repair programs. Agat’s technique, and several of its successors, work by equalizing the time spent on distinct branches within a program. These approaches essentially seek for a trade-off between the overhead imposed upon the transformed program and the amount of leakage that they mitigate. For instance, Tizpaz-Niari et al. [2019] explicitly guarantee a user-specified maximum acceptable performance overhead. However, as stated by Wu et al., such methods deliver weak guarantees, due to the presence of hidden states at microarchitectural levels and related performance optimizations inside modern CPUs.

This thesis is concerned with the so-called *white-box* mitigations, which require intervening in the software. For an overview of *black-box* approaches, such as defenses implemented at the operating-system level, we refer the reader to the comprehensive dis-

cussion presented by Cock et al. [2014]. In contrast to previous work that only attempt to balance branches, Wu et al.’s approach consists of linearizing conditional branches, so that every instruction in the influence region of a conditional branch (see Def. 3.6) executes regardless of the secret inputs. Wu et al.’s method thus ensures operation invariance. Furthermore, Wu et al. also discuss the adoption of preloading mechanisms to mitigate cache-based leaks. They proposed a *must-hit* static analysis to identify whether a memory element is definitely in the cache or not, thus allowing to optimize the insertion of preloading code. Nonetheless, preloading is architecture dependent, for the approach is customized to the dimensions of the data cache.

We believe that Rane et al. [2015]’s work might provide equally strong guarantees as Wu et al.’s, yet such guarantees are not explicitly stated. Rane et al.’s approach consists of introducing *decoy* paths, so that the adversary’s view of the program execution becomes the same for different inputs — i.e. the transformed program is operation invariant. Nevertheless, these methods still suffer from a number of shortcomings that justify the developments in this thesis. In particular, albeit both Wu et al. and Rane et al. provide arguments about the absence of side channels, they do not discuss the issue of memory safety (§1.3), a problem present in both techniques.

In 2019, Cauligi et al. [2019] proposed a *domain-specific language* called FaCT, whose focus is to write constant-time cryptographic code. The FaCT compiler uses a secrecy type system to automatically transform potentially timing-sensitive high-level code into LLVM bitcode that satisfies the constant-time constraint. FaCT is designed to be embedded into existing cryptographic projects, instead of being used as a standalone language. FaCT programs are, by design, *publicly safe*, a concept that we revisit in Definition 5.5 (§5.4.3). In the words of Cauligi et al.: “for a program to be amenable to constant-time compilation, the source must be publicly safe: it must be free from buffer overflows and undefined behavior using only public-visible information, i.e. the code must be safe even after removal of secret-dependent control-flow”. The notion of public safety is embedded into FaCT’s type system. As we shall see in §5.4.3, if a program is publicly safe, then the program resulted from our transformation will be data invariant (Theorem 5.3). In addition, our approach always guarantees operation invariance (Theorem 5.2). Therefore, for publicly-safe codes, the programs resulted from our transformation meet the same guarantees delivered by the FaCT compiler.

In 2021, we published a paper proposing a static code transformation that guarantees operation invariance while still ensuring memory safety; that is, the transformation does not add any out-of-bounds accesses to the repaired program that do not occur in the original version [Soares and Pereira, 2021]. This method, however, only works for programs whose loops have bounds known at compilation time (i.e. are fully unrollable). As mentioned in §1.5, such a technique is the topic of §4. In §5, we improve our loop-free transformation to deal with unbounded loops. When dealing with loop-free programs

where all branches are tainted, these techniques (§4 and §5) are equivalent.

Still in 2021, Borrello et al. [2021] proposed an approach to eliminate time-based side channels that combine static and dynamic analyses. Borrello et al.’s work is more general than previous work [Wu et al., 2018; Soares and Pereira, 2021], in the sense that it is capable of transforming unbounded loops. Loops are linearized *just-in-time* by replacing the normal trip count of a loop with a special induction variable that dictates how many times that loop should execute. This kind of transformation requires loop profiling in order to identify the number of iterations that the loop performs, which implies an important limitation: it is undecidable to find inputs to exercise specific parts of a program’s code (Rice’s Theorem [Rice, 1953]). In addition to *control-flow* linearization, Borrello et al. also discuss *data-flow* linearization, which consists of “obliviously accessing all the locations that the original program can possibly reference for any initial input”. To make data-flow linearization practical, they conduct a context-sensitive points-to analysis, which requires aggressive function cloning. To the best of our knowledge, Borrello et al.’s code transformation is memory safe. It is worth noting that, even though we discuss the property of data invariance throughout this thesis, the transformation we are proposing is focused on operation-based leaks; that is, we do not propose any specific kind of technique to protect against cache-based attacks.

2.2.3 Constant-Time Preservation

Implementing a constant-time policy, either by hand or through means of an automated method, does not guarantee that such a policy will remain valid on the binary level. This is because compilers typically apply several transformations onto the input code and some of them might destroy properties that are present in previous phases of the compilation [Barthe et al., 2018; Deng and Namjoshi, 2017, 2018; Besson et al., 2019]. In the context of information leak, [Barthe et al., 2018] proposed a general method for proving that a compiler optimization preserves the constant-time property of a program. For that, they defined the notion of constant-time simulation, which adapts the notion of simulation from compiler verification [Leroy, 2009]. Barthe et al. [2019] then presented a modified version of `CompCert` [Leroy, 2009] — a *formally-verified C* compiler — that succeeds in preserving the constant-time policy during compilation. Nevertheless, Barthe et al.’s verification method does not account for quantitative analysis of side-channel leakage; rather, it deals only with equality of leakage.

Besson et al. [2019] propose the notion of Information-Flow Preserving code transformation that aims at ensuring that a target program does not leak more information

than the corresponding source code. They consider an attacker who is granted physical memory access at specific observation points. Furthermore, the attacker is parameterized by the amount of information he is allowed to read: an attacker observing n bits of information during the execution of a target (transformed) program does not get an advantage over an attacker observing the same amount of bits during the execution of the source program. More recently, Barthe et al. [2021] proposed the notion of *structured leakage*, which differs from the usual modeling of leakage as sequence of observations in the sense that it is aligned with the operational semantics of programs. Similarly to Barthe et al. [2019], Barthe et al. [2021] demonstrated the application of structured leakage in the context of the Cryptographic Constant-Time property.

Chapter 3

Preliminaries

In this chapter, we introduce important notions that we shall be referring to throughout this thesis. In §3.1, we define standard concepts related to (control-flow) graphs, such as dominance and post dominance, control and data dependency, and natural loops. We close this chapter by introducing, in §3.2, the theory of Quantitative Information Flow (QIF). QIF is an emerging field that aims to explain what information leakage is and how it can be assessed *quantitatively* [Alvim et al., 2020]. As such, it is strongly correlated to the context of side-channel resistance, our topic of interest.

3.1 Graph Definitions

Partial Control-Flow Linearization [Moll and Hack, 2018] works on programs containing loops; however, it requires said loops to be *natural*. Although a natural loop is a well-established concept [Appel, 1997], Definition 3.1 revisits it for the sake of completeness. Throughout this thesis, we shall adopt the same terminology used in the LLVM documentation¹ when referring to *natural loops*.

Definition 3.1 (Natural Loop). A control-flow Graph (CFG) is a directed graph with an entry node *start*. If G is a CFG, then a loop $L \subseteq G$ is a strongly connected subgraph of G . L is called *natural* if it contains a *header* h such that every path from *start* to any node $v \in L$ goes through h .

As a consequence of Definition 3.1, the header dominates all nodes in the loop (see Def. 3.4). Most loops in programs will be natural: they are produced by statements like `while`, `do-while`, `for` and `foreach`. The creation of non-natural loops usually requires abusing the go-to statement. The original description of PCFL also requires loops to have unique *latches* (see Definition 3.2). This requirement can be met for any program via a standard compiler transformation, which we shall not explain further.

¹<https://llvm.org/docs/LoopTerminology.html>

Definition 3.2 (Loop Terminology). A *Forward Edge* is an edge from a node outside the loop to the loop header. A *Back Edge* is an edge from a node inside the loop to the loop header. A *Latch* is the source of a back edge. An *Exiting Edge* is an edge from inside the loop to a node outside it. The source of an exiting edge is called an *Exiting Block*. Similarly, the destination of an exiting edge is called an *Exit Block*.

Definition 3.3 introduces Ferrante et al. [1987]’s notion of *data* and *control dependency*. We rely on the concept of dependencies to determine whether a variable is tainted or not (Definition 5.2). Furthermore, both data and control dependency are pre-requisites for the definition of two properties: shadow and public safety (Def. 5.5).

Definition 3.3 (Data & Control Dependency). Following Ferrante et al. [1987], we say that variable y is *data dependent* on variable x if y is assigned in an instruction that uses x . Similarly, we say that a variable x is *control dependent* on a variable p if the value assigned to x depends on the outcome of a branch whose condition uses p .

Dominance and post dominance are important concepts in compiler theory [Allen, 1970; Ferrante et al., 1987; Cytron et al., 1989], and as such will be extensively mentioned throughout this text. Therefore, for the sake of completeness, Definitions 3.4 and 3.5 revisit them. Definition 3.6 relies on the notion of post dominance to define the *influence region* of a node. Influence regions will be used later on, in §5.4.3, to determine which instructions of a program must be rewritten.

Definition 3.4 (Dominance). Given a directed graph G with a unique root vertex s , we say that vertex u *dominates* vertex v if every path from s to v must go through u . Node u is the *immediate* dominator of v if, and only if, u strictly dominates v (i.e. $v \neq u$) and, for any other u' that dominates v , either u' dominates u or $u' = u$.

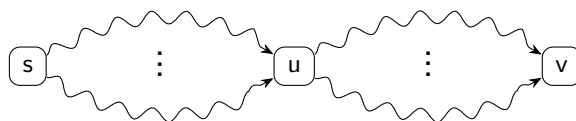


Figure 3.1. Dominance relation: every path from s — the unique start node — to node v must go through node u ; that is, u dominates v .

Definition 3.5 (Post Dominance). Given a directed graph G with a unique exit vertex d , we say that v *post-dominates* vertex u if every path from u to d must go through v . Node v is the *immediate* post dominator of u if, and only if, v strictly post-dominates u and, for any other v' that post-dominates u , either v' post-dominates v or $v' = v$.

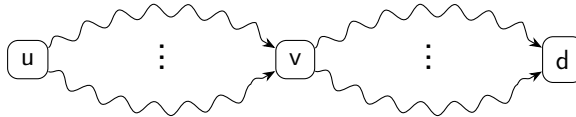


Figure 3.2. Post-dominance relation: every path from node u to d — the unique exit node — must go through node v ; that is, v post-dominates u .

Definition 3.6 (Influence Region). The influence region of a node u is the set of all nodes in paths from u to its immediate post dominator v , excluding u and v .

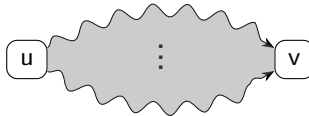


Figure 3.3. The influence region of node u is composed by all node in paths from u to its post dominator v ; that is, all nodes in the colored region.

As we shall see in §5.1, Moll and Hack [2018]’s partial control-flow algorithm depends on a special ordering called *compact topological ordering*. In short, the compact topological ordering of a graph is a topological sort in which all vertices are compact with respect to the dominance sets and the loops of the said graph (see Definition 5.1). Definition 3.7 revisits the notion of a topological sort.

Definition 3.7 (Topological Sort [Cormen et al., 2009]). A topological sort of a directed *acyclic* graph G is a linear ordering of its vertices such that, for every edge $u \rightarrow v$ in G , u comes before v in the said ordering.

3.2 Quantitative Information Flow

The theory of Quantitative Information Flow (QIF) aims to explain what information leakage is, how it can be estimated quantitatively and how to construct systems that satisfy information-flow guarantees [Alvim et al., 2020, §1]. Secrecy, in this context, is defined with regard to probabilities. That is, the knowledge an adversary has about some secret X is specified by a probability distribution π over \mathcal{X} , where π_x is the probability of each possible value x of \mathcal{X} . QIF models systems in terms of *information-theoretic channels*, which are probabilistic functions. A probabilistic system is modeled as a channel $C: \mathcal{X} \rightarrow \mathbb{D}\mathcal{Y}$, where $\mathbb{D}\mathcal{Y}$ is the set of all distributions on \mathcal{Y} . A special case is a *deterministic* channel, which maps each possible input to a single output; in this case, the channel can be described simply as $C: \mathcal{X} \rightarrow \mathcal{Y}$ [Alvim et al., 2020, §4].

Vulnerability. There are multiple ways to quantify the vulnerability associated with a secret X . One example is the *Bayes vulnerability*, which corresponds to the maximum probability that an adversary has to guess the value of X in one try. The Bayes vulnerability $V_1(\pi)$ — where the subscript “1” is chosen to reflect the “one try” aspect — is thus defined as $V_1(\pi) = \max_{x \in \mathcal{X}} \pi_x$. Nonetheless, the adversary is not necessarily restricted to a single try; in fact, the operational scenarios are infinite. QIF addresses this multiplicity of possible scenarios by parameterizing vulnerability with a gain function g that models the actions that the adversary can take, thus introducing a family of g -vulnerability measures $V_g(\pi)$ [Alvim et al., 2020, §§1,3].

Example 3.1. Suppose we roll a pair of *fair* dice: there are a total of 36 combinations, each with probability $\frac{1}{36}$ — i.e. π is a uniform distribution. This distribution π corresponds to the knowledge that the adversary has *prior* to rolling the dice. Suppose, in addition, that we only make available the sum of the values of each die. The space \mathcal{Y} of possible outputs is $\{2, \dots, 12\}$. If the result is, for instance, 4, the adversary can conclude that the only possible pairs are (1, 3), (2, 2), and (3, 1), which gives a *posterior* uniform distribution that assigns the probability $\frac{1}{3}$ to each pair. If, on the other hand, the adversary observed the result 2, then there is only one possible pair (2, 2) with probability 1. In other words, by observing the result of a system, the adversary can “update” their knowledge. Each output $y \in \mathcal{Y}$ of a channel C gives rise to a posterior distribution, meaning that the execution of a system maps a prior π into a distribution on posterior distributions, called as a *hyper-distribution* and denoted by $[\pi \triangleright C]$. The posterior g -vulnerability is denoted by $V_g[\pi \triangleright C]$ [Alvim et al., 2020, §§1, 4.3, 5.1].

From a security point of view, Example 3.1 can be seen as an adversary that wants to discover a two-digit password [Alvim et al., 2020, §1.1.1]. In our context, however, we are interested in adversaries that try to obtain sensitive data by observing information that flows through *time-based side channels* instead of the usual output of the system. Consider, for example, functions **oFdF** and **oTdT** from, respectively, Figures 1.1 (a) and (d). These two procedures implement a password checker. The first one is neither operation nor data invariant, because the sequences of operations and memory accesses depend on the value of the secret input **pw**. In contrast, the second is both operation and data invariant. To illustrate, let $\mathbf{n} = 1$, $\mathbf{g} = [0]$ and $|\mathbf{pw}| = 1$. Furthermore, let, under such assumptions, $A^{\mathbf{n}=1, \mathbf{g}=[0]}$ and $D^{\mathbf{n}=1, \mathbf{g}=[0]}$ be the deterministic channels that correspond to the trace of operations relative to the execution of, respectively, **oFdF** and **oTdT**. Example 3.2 shows the Bayes vulnerability of the two channels, assuming a uniform prior distribution π for the secret input $\mathbf{pw} = [n], 0 \leq n \leq 9$.

Example 3.2. The prior Bayes vulnerability is the maximum of the prior distribution π ; that is, $V_1(\pi) = \frac{1}{10}$. Figures 3.4 (a) and (b) show the partition induced by, respectively, channels $A^{\mathbf{n}=1, \mathbf{g}=[0]}$ and $D^{\mathbf{n}=1, \mathbf{g}=[0]}$. Channel $A^{\mathbf{n}=1, \mathbf{g}=[0]}$ maps the inputs $\mathbf{pw} = [0]$ and

$\mathbf{pw} = [i]$, $1 \leq i \leq 9$, to two distinct traces, each one with probability 1 (recall that the channels are deterministic). Assuming that the adversary observed the trace relative to $\mathbf{pw} = [0]$, they now know — with probability 1 — the secret input. The probability of observing that trace is $\frac{1}{10}$, given that $A^{n=1, \mathbf{g}=[0]}$ maps only one input to such a trace. On the other hand, for $\mathbf{pw} = [i]$, $1 \leq i \leq 9$, the posterior distribution assigns a probability of $\frac{1}{9}$ to each possible i , and the probability of observing the corresponding trace is $\frac{9}{10}$. The posterior Bayes vulnerability is the expected value of the Bayes vulnerability over the hyper-distribution, i.e.

$$\begin{aligned} V_1 [\pi \triangleright A^{n=1, \mathbf{g}=[0]}] &= \frac{1}{10} V_1 \left(\frac{1}{1} \right) + \frac{9}{10} V_1 \left(\frac{1}{9}, \dots, \frac{1}{9} \right) \\ &= \frac{1}{10} + \frac{1}{10} \\ &= \frac{1}{5} \\ &> V_1(\pi). \end{aligned}$$

Notice that the posterior vulnerability is larger than the prior. This reflects the fact that the result of channel $A^{n=1, \mathbf{g}=[0]}$ helps the adversary to choose the best action for them. Let us now analyze channel $D^{n=1, \mathbf{g}=[0]}$. Unlike channel $A^{n=1, \mathbf{g}=[0]}$, $D^{n=1, \mathbf{g}=[0]}$ maps every possible input \mathbf{pw} to a unique trace. Such a trace is observed with probability 1, and the posterior distribution is a uniform distribution $(\frac{1}{10}, \dots, \frac{1}{10})$. Therefore, the Bayes vulnerability of channel $D^{n=1, \mathbf{g}=[0]}$ is

$$\begin{aligned} V_1 [\pi \triangleright D^{n=1, \mathbf{g}=[0]}] &= 1 V_1 \left(\frac{1}{10}, \dots, \frac{1}{10} \right) \\ &= \frac{1}{10} \\ &= V_1(\pi), \end{aligned}$$

which, as demonstrated, is exactly the prior Bayes vulnerability $V_1(\pi)$. That is, the adversary cannot increase their knowledge by observing the output of $D^{n=1, \mathbf{g}=[0]}$.

Refinement. In this work, we are interested in deterministic programs. Thus, we shall focus on deterministic channels. A crucial constraint that we wish our transformation to satisfy is that no program produced by our technique leaks more than its original counterpart. This is the concept of *refinement*, and it is extensively described in Alvim et al. [2020]. There are two important formulations of refinement: *structural* refinement, which is how refinement is “implemented” by the developer, and *testing* refinement, which corresponds to the customer’s point of view. The latter is harder to determine, but gives more information with regard to security: as Definition 3.8 shows, its formulation relies

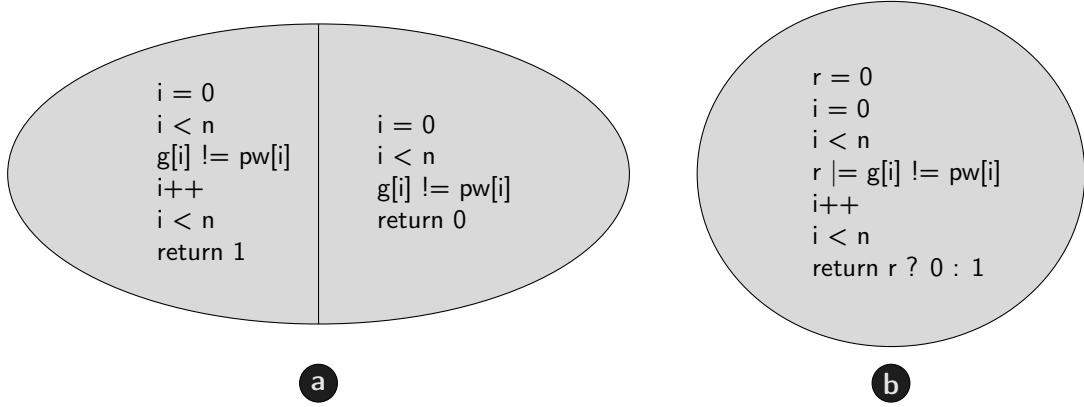


Figure 3.4. (a) Partition induced by $A^{n=1, g=[0]}$, which corresponds to function oFdF from Figure 1.1 (a). (b) Partition induced by $D^{n=1, g=[0]}$, which corresponds to function oTdT from Figure 1.1 (b). Channel $A^{n=1, g=[0]}$ is refined by $D^{n=1, g=[0]}$, i.e. $D^{n=1, g=[0]}$ leaks no more than $A^{n=1, g=[0]}$.

explicitly on g -vulnerability, indicating whether a channel is less vulnerable than another. The former gives no information with regard to vulnerability, but it is easier to compute. As such, we shall focus on the definition of *structural* refinement (deterministic case). Nevertheless, as demonstrated by Alvim et al. [2020, Theorem 9.13], both formulations are, in fact, exactly the same.

Definition 3.8 (Testing Refinement ($\sqsubseteq_{\mathbb{G}}$) [Alvim et al., 2020, Def. 9.10]). Given channels C and D , over the same input \mathcal{X} , we say that C is testing-refined by D , written $C \sqsubseteq_{\mathbb{G}} D$, if for any prior π and gain function $g: \mathbb{G}\mathcal{X}$ we have $V_g[\pi \triangleright C] \geq V_g[\pi \triangleright D]$.

Any deterministic channel $C: \mathcal{X} \rightarrow \mathcal{Y}$ induces a *partition* on \mathcal{X} , which is a set of mutually disjoint subsets of \mathcal{X} called *cells*. Two inputs $x_1, x_2 \in \mathcal{X}$ belong to the same cell just when $C(x_1) = C(x_2)$. As shown in Figure 3.4 and explored in Example 3.2, channels $A^{n=1, g=[0]}$ and $D^{n=1, g=[0]}$ partition the space of the secret input pw into, respectively, two and one cells. As Definition 3.9 demonstrates, we can use the partition induced by deterministic channels to determine which of them is more secure. Example 3.3 applies this notion to channels $A^{n=1, g=[0]}$ and $D^{n=1, g=[0]}$. From now on, we write $\langle \dots \rangle$ to denote ordered sequences.

Definition 3.9 (Structural Refinement (\sqsubseteq_{\circ}) [Alvim et al., 2020, Def. 9.1]). Two *deterministic* channels C and D on input \mathcal{X} are said to be in the *structural-refinement* relation, written $C \sqsubseteq_{\circ} D$, just when the partition induced by D is coarser than that induced by C , in that each of D 's cells is formed by *merging* one or more of C 's cells.

Example 3.3. $A^{n=1, g=[0]}$ maps the input $\text{pw} = [0]$ to the trace of operations

$$\langle i = 0, i < n, g[i] \neq \text{pw}[i], i++, i < n, \text{return } 1 \rangle.$$

Similarly, channel $A^{n=1, g=[0]}$ maps $\text{pw} = [i]$, $1 \leq i \leq 9$, to

$\langle i = 0, i < n, g[i] \neq \text{pw}[i], \text{return } 0 \rangle$.

In contrast, channel $D^{n=1, g=[0]}$ maps every possible value of the secret input pw to

$\langle r = 0, i = 0, i < n, r \mid= g[i] \neq \text{pw}[i], i++, i < n, \text{return } r ? 0 : 1 \rangle$.

This single cell induced by $D^{n=1, g=[0]}$ comprises all the inputs that $A^{n=1, g=[0]}$ maps to two distinct cells. Thus, we can think of the unique cell induced by $D^{n=1, g=[0]}$ as the merge of the two cells induced by $A^{n=1, g=[0]}$, meaning that $A^{n=1, g=[0]} \sqsubseteq_{\circ} D^{n=1, g=[0]}$. In other words, function oTdT , from Figure 1.1 (d), is more secure than procedure oFdF , from Figure 1.1 (a), at least with respect to instruction-based leaks.

Extreme channels $\mathbb{0}$ and $\mathbb{1}$. There are two particular channels that deserve their own discussion space: the channel that leaks *nothing* and the channel that leaks *everything*. They are extreme opposites of each other. The former is denoted as channel $\mathbb{1}$: it is the channel that *preserves* secrecy. In contrast, the latter is denoted as $\mathbb{0}$: it is the channel that *annihilates* secrecy [Alvim et al., 2020, §4]. Ideally, our transformation should produce a program corresponding to channel $\mathbb{1}$. This, however, is not always possible. Nevertheless, as we shall see later in §5.4.3, there is a special class of programs for which our transformation always guarantees the “no leakage” property; these are the *publicly safe* programs (Definition 5.5).

Chapter 4

Transforming Loop-Free Programs

In this chapter, we will describe a transformation that targets loop-free programs. For now, we shall consider every branch as tainted. Then, in §5, we will extend this transformation to deal with general loops, using Moll and Hack [2018]’s partial control-flow linearization. Therefore, we leave formal results on correctness and side-channel resistance for §5. We start by introducing the baseline language that we shall use throughout the entire thesis to explain our ideas (§4.1). Then, we present a static analysis to compute the conditions that control the execution of each basic block (§4.2). In §4.3, we develop the intraprocedural transformation framework. Finally, in §4.4, we discuss how to expand this transformation to an interprocedural approach.

4.1 Baseline Language

Figure 4.1 shows the syntax of the toy language that will be used to explain our ideas. In Figure 4.1, $\{\}$ indicates zero or more occurrences, $[]$ denotes optional terms, id represents names of variables, n stands for numerals, and ℓ ranges over basic block labels. In this thesis, we assume all programs to be in the *Static Single Assignment* (SSA) form [Cytron et al., 1989].¹ Thus, every variable has a single definition site and the definition of a variable dominates all its uses. To meet these properties, the toy language is equipped with **phi** functions — special instructions that join multiple definitions of the same variable. In addition, the language provides a **ctsel** (constant-time selector) operation, which is parameterized by a condition c , such that $\mathbf{ctsel} c, v_t, v_f \equiv v_t$ if $c \equiv \mathbf{true}$ or v_f otherwise.² We represent stores with a left arrow instead of an equal sign to distinguish them from simple assignments. For convenience, we write stores of the form $v[0] \leftarrow x$ as $v \leftarrow x$. We shall henceforth write $\mathbf{inst} @ \ell$ to indicate that \mathbf{inst} belongs to

¹The SSA assumption is not a necessary condition to enable the code transformation described in this thesis. Nevertheless, we assume it for convenience, because this format is adopted in the LLVM program representation.

²We treat zero as false and any integer different from zero as true.

a block labeled by ℓ . Example 4.1 shows a program in our toy language.

$$\begin{aligned}
 \text{Program} &::= \{ \text{BasicBlock} \} \\
 \text{BasicBlock} &::= \ell: \{ \text{Assignment} \} \text{Terminator} \\
 \text{Assignment} &::= id = \mathbf{public} \\
 &\quad | id = \mathbf{secret} \\
 &\quad | id = Expr \\
 &\quad | id = id \text{'[Value ']} \\
 &\quad | id \text{'[Value ']} \leftarrow Expr \\
 &\quad | id = \mathbf{phi} \text{'[Expr, } \ell \text{']} \{ , \text{'[Expr, } \ell \text{']} \} \\
 &\quad | id = \mathbf{ctsel} \text{ Value, Value, Value} \\
 \text{Terminator} &::= \mathbf{br} [\text{Value, } \ell,] \ell | \mathbf{halt} \\
 \text{Expression} &::= \text{Value} | \mathit{unop} \text{ Value} | \text{Value } \mathit{binop} \text{ Value} \\
 \text{Value} &::= \mathbf{true} | \mathbf{false} | n | id
 \end{aligned}$$

Figure 4.1. Syntax of the baseline language used throughout the thesis to design the isochronous transformation.

Example 4.1. Figure 4.2 shows the implementation of function `oFdF`, from Figure 1.1 (a), in our toy language. Function `oFdF` compares a user’s guess `g` with the secret password `pw`. It immediately returns `false` if the test at line 6 evaluates to `true`; else, it returns `true`, meaning that `g` equals `pw`. Hence, `oFdF` might leak the secret password due to the non-constant behavior of the loop at lines 5–9.

4.2 Predication

The *observable* effects of a program are the set of state modifications that said program carries out on the machine that it controls. In the context of this work, observable effects are memory writes. If P is a program and P_l is the linearization of P , then we want both P and P_l to have the same observable effects when given the same inputs — in this chapter, we are not distinguishing public and secret inputs yet. To achieve this property, we need to ensure that instructions of P_l only cause observable effects when their counterparts in P do. For that, we resort to *predication*, a classic compiler transformation. To predicate instructions, we rely on the notion of *edge* and *block conditions*. We start by formalizing these concepts in Definition 4.1 for *loop-free* programs. Example 4.2 illustrates these concepts.

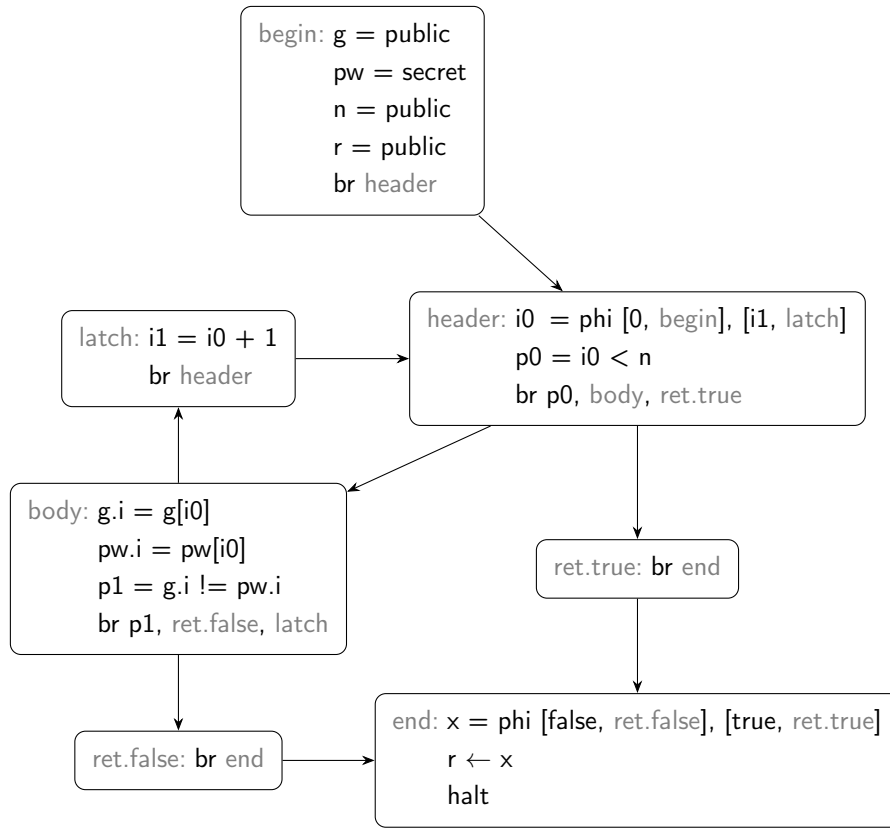


Figure 4.2. Control-flow graph of the password comparison function `oFdF` seen in Figure 1.1 (a), written in our baseline language.

Definition 4.1 (Block & Edge Conditions). The block condition $BC(v)$ determines when block v executes. The edge condition $EC(u \rightarrow v)$ determines when edge $u \rightarrow v$ is traversed. The equations in Figure 4.3 define these mutual relations.

$$\frac{\text{terminator}(\ell) = \text{br } p, \ell', _}{EC(\ell \rightarrow \ell') = BC(\ell) \wedge p}$$

$$\frac{\text{terminator}(\ell) = \text{br } p, _ , \ell'}{EC(\ell \rightarrow \ell') = BC(\ell) \wedge \bar{p}}$$

$$\frac{\text{terminator}(\ell) = \text{br } \ell'}{EC(\ell \rightarrow \ell') = BC(\ell)}$$

$$\frac{\text{predecessors}(\ell) = \{\ell_1, \dots, \ell_n\}}{BC(\ell) = \bigvee_{j=1}^n EC(\ell_j \rightarrow \ell)}$$

Figure 4.3. Recursive definitions of block and edge conditions. Function $\text{predecessors}(\ell)$ gives all the blocks that are predecessors of block ℓ . Function $\text{terminator}(\ell)$ returns the last instruction of the block labeled by ℓ (see Figure 4.1).

Example 4.2. Figure 4.4 shows the CFG of a simplified version of function `oFdF` seen in Figure 4.2, with $n = 2$ and the loop unrolled. Each edge is labeled with its corresponding edge condition. Block `if.0` always executes; hence, its block condition is `true`. Block

`if.1` executes whenever `p0` is false; thus, its block condition is $\overline{p0}$. The block condition of `end` is the disjunction $(p0 \vee (\overline{p0} \wedge p1)) \vee (\overline{p0} \wedge \overline{p1})$, which reduces to `true`, since the last basic block `end` always executes.

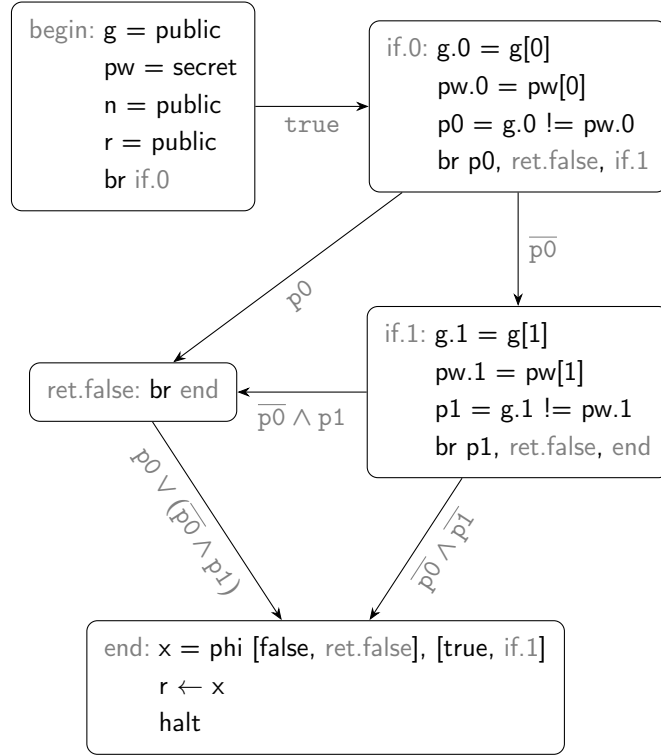


Figure 4.4. CFG of function `oFdF` from Figure 4.2, with $n = 2$ and the loop unrolled. Edges are labeled with edge conditions (Definition 4.1, Figure 4.3).

4.3 Rewriting System

4.3.1 Control Flow

To ensure operation invariance, a property that we formalize later in §5.4.1, we must linearize the control-flow graph of the program. Recall that, in this chapter, we are considering every branch as tainted. Thus, our linearization algorithm must delete all branches of the program. Figure 4.5 shows the rule that we use to rewrite these branches. Function $rewrite_{br}$ takes as input the original CFG G and a terminator `br ... @ ℓ` , which can be both conditional or unconditional, and replaces it with an unconditional branch

linking the basic block ℓ with its successor ℓ' in the topological order (see Def. 3.7) of the CFG. Notice that we are dealing with acyclic graphs; thus, we can safely rely on topological sorting. Example 4.3 illustrates the linearization process.

$$\frac{\text{topological}(G) = \langle \ell_1, \dots, \ell_i, \ell_j, \dots, \ell_n \rangle \quad \ell_i = \ell}{\text{rewrite}_{br}(\mathbf{br} \dots @ \ell, G) = br \ell_j}$$

Figure 4.5. Transformation rule for linearizing the CFG of a program. Function rewrite_{br} replaces every branch with another branch whose target is the next node in the topological order. Function $\text{topological}(G)$ returns the topological sort (Definition 3.7) of the acyclic graph G .

Example 4.3. Figure 4.6 shows the linearization of the CFG early seen in Figure 4.4, considering the following topological sort of the basic blocks: `begin`, `if.0`, `if.1`, `ret.false`, `end`. Notice that the conditional branches at blocks `if.0` and `if.1` no longer exist. In other words, the linearized program will execute all instructions regardless of any input. Therefore, the linearized code is operation invariant.

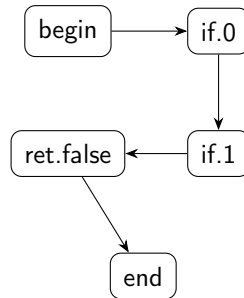


Figure 4.6. Linearization of the CFG from Figure 4.4, following the rewrite rule defined in Figure 4.5. The topological sort adopted for this transformation was: `begin`, `if.0`, `if.1`, `ret.false`, `end`.

4.3.2 Phi Functions

As discussed in §4.3.1, in this chapter we are linearizing programs entirely. Therefore, in the linearized program P_l , every basic block has at most one predecessor, meaning that every phi function with at least two incoming values become invalid. Hence, we must rewrite them. Figure 4.7 defines the transformation rules that we use in this process. Example 4.4 shows the application of such rules into the linearized version of the program seen in Figure 4.4.

$$\frac{}{\text{rewrite}_\phi(\mathbf{x} = \mathbf{phi}[e_1, \ell_1], [e_2, \ell_2]) = (\mathbf{x} = \text{ctsel } BC(\ell_1), e_1, e_2)}$$

$$\frac{\text{rewrite}_\phi(\mathbf{y} = \mathbf{phi}[e_2, \ell_2], \dots, [e_n, \ell_n]) = (\mathbf{y} = \text{ctsel } \dots)}{\text{rewrite}_\phi(\mathbf{x} = \mathbf{phi}[e_1, \ell_1], [e_2, \ell_2], \dots, [e_n, \ell_n]) = (\mathbf{x} = \text{ctsel } BC(\ell_1), e_1, \mathbf{y})}$$

Figure 4.7. Transformation rules used to rewrite phi functions, pre-PCFL. Function rewrite_ϕ takes a phi node and returns a combination of `ctsel`s that are equivalent to that phi node. If the phi node has only one incoming value, it is equivalent to a simple assignment, and thus do not need any modifications.

Example 4.4. The phi function at block `g` in Figure 4.8 (a) is rewritten in Figure 4.8 (b). Notice that the original edges $d \rightarrow g$ and $f \rightarrow g$ were deleted in (b). Hence, the phi node became invalid. To fix that, the phi node was replaced by a combination of two `ctsel`s parameterized by the edge conditions of the original edges $d \rightarrow g$ and $e \rightarrow g$, which in Figure 4.8 are encoded as, respectively, variables `ec.d_g` and `ec.e_g`.

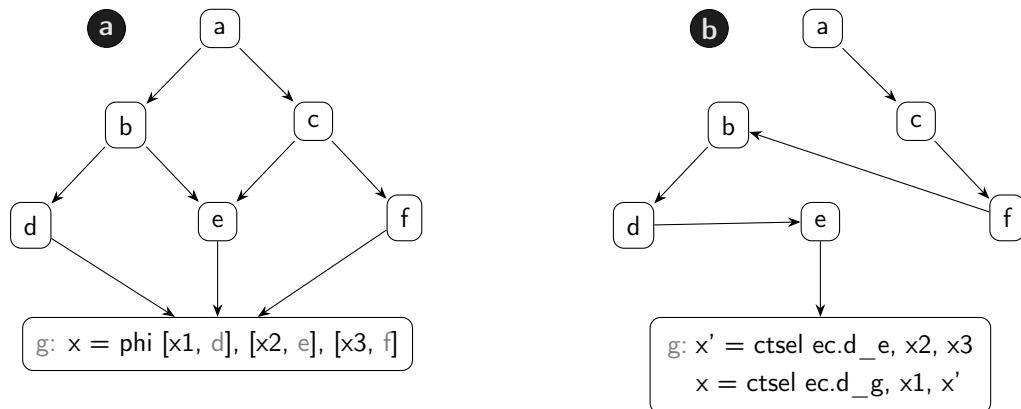


Figure 4.8. (a) CFG before linearization. (b) CFG after linearization, with the phi function at block `g` rewritten according to Figure 4.7; variables `ec.d_g` and `ec.e_g` store the edge condition of, respectively, edges $d \rightarrow g$ and $e \rightarrow g$ from the original graph (see Definition 4.1, Figure 4.3).

4.3.3 Memory Operations

The transformation of memory operations has two goals. First, to guarantee that stores only produce observable effects in the linearized program P_l when their counterpart would cause observable effects in the original program P . Second, to ensure memory safety, a concept that we formalize with Definition 4.2. Figure 4.9 introduces the rules that

we use to rewrite loads and stores. To identify which instructions require interventions, we rely on the *influence region* (Def. 3.6) of basic blocks. We thus apply the rules from Fig. 4.9 to operations in the influence region of any block that ends with a conditional branch. The next example depicts the influence region of such a block.

Example 4.5. The influence region of the basic block `if.0`, from Figure 4.2, is the set formed by blocks in paths from `if.0` to its post-dominator `end: if.1` and `ret.false`. Given that `if.0` ends with a conditional branch, every memory operation in its influence region must be rewritten. Since `if.0` itself is not within the influence region of any conditional branch, the memory operations that belong to `if.0` can be kept without any changes. This, in fact, reflects the behavior of the original program: the two loads at `if.0` always execute, regardless of the inputs.

Definition 4.2 (Memory-Safe Transformation). Let $T: Program \rightarrow Program$ be a transformation over programs. Then, T is said to be a memory-safe transformation if, and only if, for every program P it follows that $T(P)$ does not contain any out-of-bounds memory accesses that do not occur in P .

$$\frac{c = i < size(\mathbf{x}) \quad c' = BC(\ell) \parallel c \quad i' = \mathbf{ctsel} \ c', i, 0 \quad \mathbf{a} = \mathbf{ctsel} \ c', \mathbf{x}, \mathbf{shadow}}{rewrite_{ld}(\mathbf{y} = \mathbf{x}[i] @ \ell) = (\mathbf{y} = \mathbf{a}[i'], \mathbf{a}, i')}$$

$$\frac{rewrite_{ld}(\mathbf{z} = \mathbf{y}[i] @ \ell, G) = \{\mathbf{z}, \mathbf{a}, i'\} \quad \mathbf{x}' = \mathbf{ctsel} \ BC(\ell), x, \mathbf{z}}{rewrite_{st}(\mathbf{y}[i] \leftarrow x @ \ell) = \mathbf{a}[i'] \leftarrow \mathbf{x}'}$$

Figure 4.9. Transformation rules for memory operations. Function $rewrite_{ld}$ makes loads memory safe, while function $rewrite_{st}$ makes stores both memory safe and sound with respect to whether they should or not take effect. Both rules rely on block conditions (see Definition 4.1).

Loads. Function $rewrite_{ld}$, in Figure 4.9, takes the original load and returns a new load that is memory safe, along with the base address and the index that compose the new access. For that, we replace memory accesses that should not occur — i.e. the block condition is false — and are not safe with accesses to a *shadow* address. To determine whether an access is safe or not, we need the size of the structure being manipulated. This can be obtained in multiple ways, e.g. by inferring the size or by asking the user to provide it; in Figure 4.9, we abstract away this computation by relying on a function named *size*. If the size of the value cannot be determined, the access is still guaranteed to be safe, but it becomes data variant (see Theorem 5.3). In practice, we conservatively estimate the size of LLVM arrays without user intervention.

Example 4.6. Consider the load `pw.1 = pw[1]` in Figure 4.4. Let `bc.if.1` store the block condition of node `if.1` and `pw.size` store the size of the input `pw`. Then, following function *rewrite_{ld}* from Figure 4.9, we have the transformation that takes the original code in Figure 4.10 (a) to produce the rewritten code in Figure 4.10 (b).

<p>Ⓐ <code>pw.1 = pw[1]</code></p>	<p>Ⓒ <code>pw[1] ← x</code></p>
<p>Ⓑ <code>c = 1 < pw.size</code> <code>c' = bc.if.1 c</code> <code>i = ctsel c', 1, 0</code> <code>a = ctsel c', pw, shadow</code> <code>pw.1 = a[i]</code></p>	<p>Ⓓ following load seen in part (b): <code>x' = ctsel bc.if.1, x, pw.1</code> <code>a[i] ← x'</code></p>

Figure 4.10. (a) Original load from Figure 4.4. (b) Transformed load. (c) Example of a store. (d) Transformed store.

Stores. Function *rewrite_{st}*, in Figure 4.9, takes the original store and produces a new store that is both memory safe and sound with respect to observable effects. We first create a *safe* load to get the current value stored in that memory region (or in the shadow memory, depending on the circumstances). Then, we use the block condition $BC(\ell)$ to select between the current value and the value to be stored: if $BC(\ell)$ is true, the original store is performed, updating the value under that address and producing an observable effect; otherwise, the store is silent.

Example 4.7. Suppose that we had a store like `pw[1] ← x` in block `if.1` in Figure 4.2 (b). Following function *rewrite_{st}* from Figure 4.9, we first create a safe load, as shown in Example 4.6. For convenience, let us reuse `pw.1`. The store will then be rewritten from the original code seen in Figure 4.10 (c) into the sequence in Figure 4.10 (d).

4.3.4 Final Example

Figure 4.11 shows the transformed version of the code seen in Figure 4.4, which we obtain after applying onto it the techniques discussed in this chapter. Variables `g.size` and `pw.size` hold the sizes of the arrays `g` and `pw` whenever the function is invoked. If the

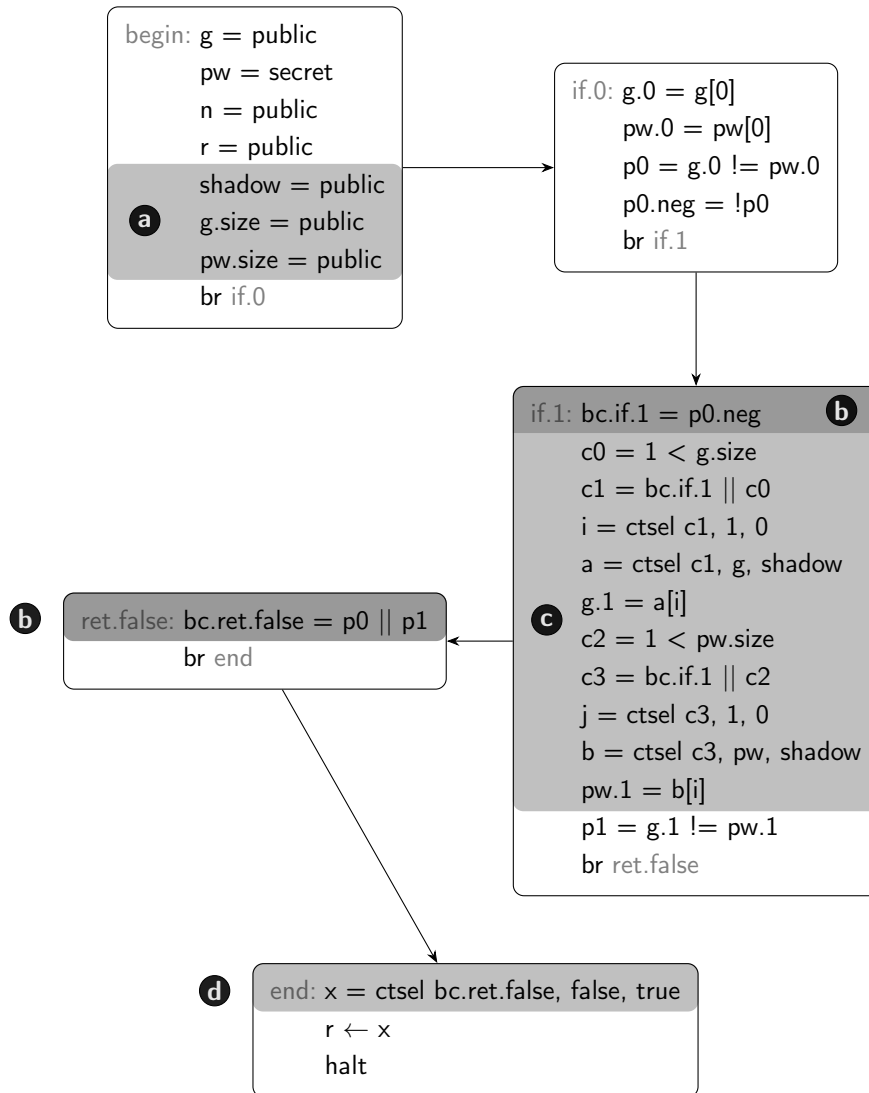


Figure 4.11. CFG from Figure 4.4 after linearization and with the instructions rewritten. (a) The shadow memory and the size of inputs g and pw , used in the transformation of loads. (b) Computation of block conditions (§4.2, Figure 4.3). (c) Predication of load instructions to ensure memory safety (§4.3.3, Figure 4.9). (d) Transformation of a phi function (§4.3.2, Figure 4.7).

length of the array is not known statically, then its size is initialized with zero. Notice that the length does not have to be a constant: it can be a symbolic expression. Whenever an expression used to index an array is, in the original program, within the influence region of a conditional branch, we compare such an expression against the length of that array. This is the case of the loads `g[1]` and `pw[1]`, as we can observe in Figure 4.11 (c). If the comparison returns false and the block condition is false, the special variable `shadow` is used as a surrogate address. Finally, we replace phi functions with a combination of `ctsels`, as seen in Figure 4.11 (d)

4.4 Interprocedural Transformation

Cryptographic algorithms might be composed by a combination of functions. One way to deal with multiple functions is to inline them. This approach, however, is not always the best solution, for the transformation might result in code that is large enough to render our techniques impractical. Therefore, in this section we present a simple, yet efficient, rewriting principle that we use to avoid inlining when carrying out isochronification. Every function that is called (the callee) within transformed code (the caller) is modified in three ways:

1. The signature of the callee is augmented to receive a condition (Figure 4.12-ii);
2. The body of the callee is surrounded by a conditional test, guarded by the new condition (Figure 4.12-iii); and
3. The caller is modified so that the block condition at the invocation point is passed to the callee (Figure 4.12-i).

Notice that the conditions are computed as part of the transformation itself. In other words, the transformation just described receives, for free, the block conditions as a byproduct of the analysis discussed in §4.2 and shown in Figure 4.3.

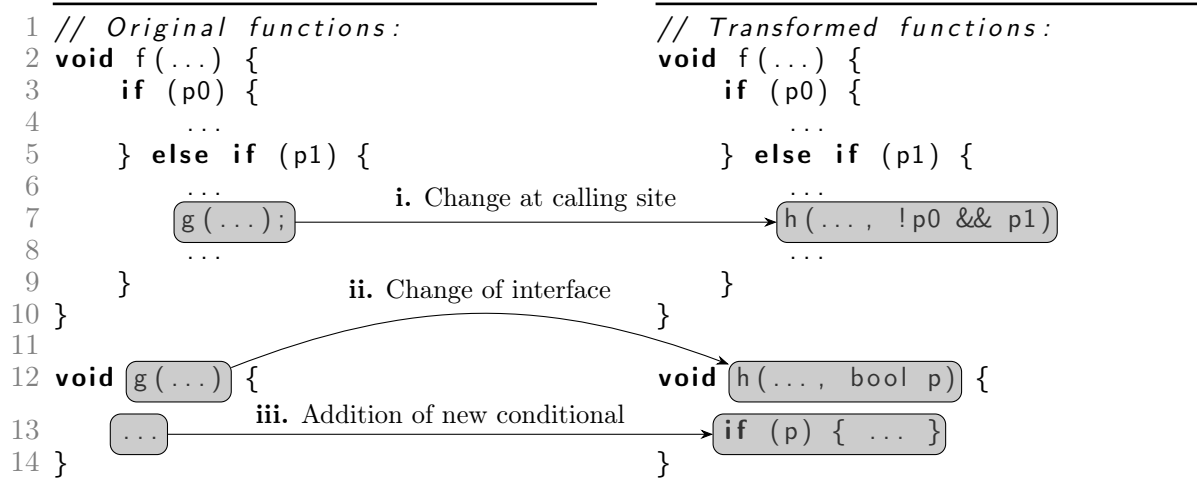


Figure 4.12. Interprocedural transformations.

Chapter 5

From PCFL to SCE

In §4, we introduced a code transformation that targets loop-free programs only. The technique discussed in §4 makes no distinction with regard to public and secret inputs. As a result, it eliminates all branches that exist in the original program. In this chapter, we will extend this transformation so that it (i) linearizes only secret-dependent branches and (ii) deals with general (unbounded) loops. We shall implement our ideas on top of the same toy language defined in §4.1. We start by presenting Moll and Hack [2018]’s partial control-flow linearization algorithm (§5.1). Then, in §5.2, we discuss how we identify which branches are tainted (i.e. must be linearized). In §5.3, we expand the static analysis that compute block and edge conditions (Definition 4.1) to handle loops. In §5.4, we develop the new rewriting system. Finally, in §5.5, we close this chapter by showing that our technique is correct.

5.1 Partial Control-Flow Linearization

Partial control-flow linearization is a code generation technique conceived for the single instruction, multiple data (SIMD) execution model. SIMD machines are an economically viable alternative to process the so-called “embarrassingly parallel” workloads. The model characterizes the stream processors in graphics processing units (GPUs) [Garland and Kirk, 2010] or the vector units found in modern CPUs [Chen et al., 2021], like Intel x86’s SSE and AVX, AMD’s 3DNow!, ARM’s NEON, Sparc’s VIS, PowerPC’s AltiVec and MIPS’ MSA. In this environment, multiple *processing elements*, or threads, simultaneously execute the same operation on different data. Example 5.1 will make this *modus operandi* more concrete.

Example 5.1. Figure 5.1 (a) shows a simplified CUDA kernel — a program meant to run on a graphics processing unit. This program counts how many occurrences of keys stored in the array `q` are present in the matrix `d`. Results are stored in the array `r`. Syntactically, function `search` looks like standard C code. Semantically, it is very different: the program

will be executed by multiple threads in lockstep. Although threads see the same input arguments q , d and r , they differ with respect to the special register tid . This identifier is unique per thread. A common pattern in this environment is to use this register to set up the work that each processing element will carry out. In this example, each thread uses its own tid as the index of the value in q that must be searched in the matrix d . The thread identifier also indicates the position in r where each thread will store its answer.

```

1  __global__ void search(int *q, int d[T][N], int *r) {
2      for (int i = 0; i < N; i++) {
3          if (q[tid] == d[tid][i])
4              r[tid] += 1
5      }
6  }

```

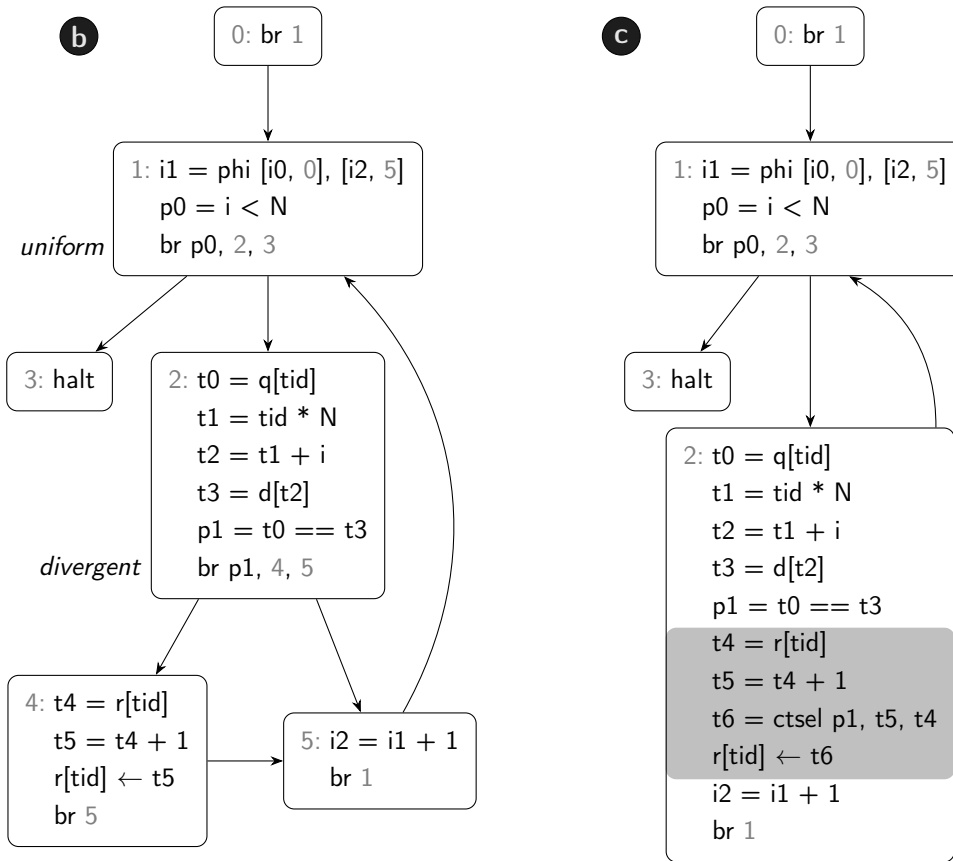


Figure 5.1. (a) CUDA kernel that counts occurrences of keys in a matrix. (b) Control-flow graph of the kernel. (c) Partially linearized control-flow graph.

The SIMD model suits very well *straight-line code*, that is, code without branches, for the execution flow of the different processing elements never diverges in this setting. However, programs do have branches over whose outcome threads might disagree. In face of divergences, threads still move in lockstep at the hardware level; however, some processing elements stop doing useful work. Typically, predicated instructions are used to

ensure that processing elements only write back their results when they run along paths actually taken within the program. Example 5.2 illustrates this trend.

Example 5.2. Figure 5.1 (b) shows the control-flow graph (CFG) of the kernel seen in Figure 5.1 (a). This CFG contains two conditional branches at the end of blocks 1 and 2. The former can be determined to be uniform, meaning that threads always take the same decision when executing it; the latter is divergent. There exist standard compiler analyses to separate uniform and divergent branches [Coutinho et al., 2011; Sampaio et al., 2014]. Figure 5.1 (c) shows a linearization of the CFG in Figure 5.1 (b) that removes the divergent branch while preserving the uniform one. The store in block 4 still happens, but is *silent*, unless the predicate `p1`, which controls the divergent branch in Figure 5.1 (b), is true. A silent store writes to memory the same value that was already there. A conditional selector (`ctsel`), guarded by `p1`, determines whether the store is silent or not.

Because only some, but not all, branches in Example 5.2 are removed, the linearization is said to be *partial*. The current state-of-the-art algorithm for partial control-flow linearization is due to Moll and Hack [2018]. Figure 5.2 shows a version of that algorithm in Python syntax. We present the algorithm for the sake of completeness, for it is extensively described in its original exposition [Moll and Hack, 2018]. The algorithm visits the basic blocks in the target graph in a special order: the *compact topological ordering*, which is formalized in Definition 5.1.

Definition 5.1 (Compact Topological Ordering). An n -sequence of vertices v_1, \dots, v_n is *dominance compact* if whenever v_1 dominates v_n then v_1 dominates every $v_i, 1 < i < n$. Similarly, an n -sequence of vertices v_1, \dots, v_n is *loop compact* if whenever v_1 and v_n belong to a loop L then every $v_i, i < i < n$, belong to L as well. A topological ordering of G is *compact* if it is both dominance and loop compact with respect to all dominance sets and loops.

Function `compact_order`, in Figure 5.2 produces a compact topological ordering “`Index`” of the vertices in graph G . This function uses an auxiliary routine, `topological_sort`, to produce some topological ordering of the nodes in a graph. Our code also assumes the existence of an “`idom`” relation, such that `idom(v, u)` is true if v is the *immediate dominator* of u . We say that v is the immediate dominator of node u if, and only if, v dominates u , and for any other node t that also dominates u , t also dominates v .

Function `linearize` in Figure 5.2 builds a graph GL that is a linearized version of the graph G . Once a compact ordering “`Index`” of the vertices in the CFG is built, the function `linearize`, in Figure 5.2 visits this sequence of nodes in order. The algorithm keeps a set D of *deferred* edges, which are edges that point to *attractors*: vertices that will attract the next nodes yet to be visited. Once attractors are connected to the linearized

```

1 def compact_order(G, entry):
2     tsort = topological_sort(G)
3     end = len(tsort)
4     def schedule(u, start):
5         bidx = [u]
6         for i in range(start, end):
7             v = tsort[i]
8             if G.idom(v, u): # v is the immediate dominator
9                 bidx += schedule(v, i + 1)
10        return bidx
11    return schedule(tsort[0], 0)
12
13 def min_index(S, indices):
14    return first(i for i in indices if i in S)
15
16 def linearize(G):
17    Index = compact_order(G, G.entry)
18    GL = Graph(G.num_vertices)
19    D = set() # The set of deferred edges
20    for b in Index:
21        T = {s for (v, s) in D if v == b}
22        if G.is_uniform(b):
23            for s in G.successors(b):
24                nxt = min_index(T + {s}, Index)
25                GL.add_edge(b, nxt)
26                D = D + {(nxt, t) for t in T + {s} \ {nxt}}
27        else: # b is divergent or is unconditional
28            S = G.successors(b)
29            if (S):
30                nxt = min_index(T + S, Index)
31                GL.add_edge(b, nxt)
32                D = D + {(nxt, t) for t in T + S \ {nxt}}
33    D = D \ {(v, s) for (v, s) in D if v == b}
34    return GL

```

Figure 5.2. Partial Control-Flow Linearization. `linearize(G)` produces a graph `GL` that is a partially linearized version of `G`. A preprocessing step removes the back edges in `G` before linearization; hence, `linearize` receives an acyclic graph.

graph, edges pointing to them are removed from `D`. The successors of uniform branches can still change (lines 23–26); however, the out-degree of those branches remains the same. Divergent branches undergo more extensive changes: they keep only one successor (lines 29–32). The links that disappear are added to the set of deferred edges; hence, these successors become attractors to be eventually reintegrated into the linearized graph.

Example 5.3. Figure 5.3 shows the order in which edges are added to the linearized graph. The original edges are mostly kept, except when node 2 is visited (`b = 2`) in line 19 of Figure 5.2. Node 2 contains a divergent branch. Thus, the node `nxt` of smallest index among the attractors and successors of 2 is chosen to bear the edge that leaves node 2 (lines 27–31 in Figure 5.2). The other successors `s` of node 2 are marked as targets of edges `nxt` \rightarrow `s` in the set `D` of deferred edges.

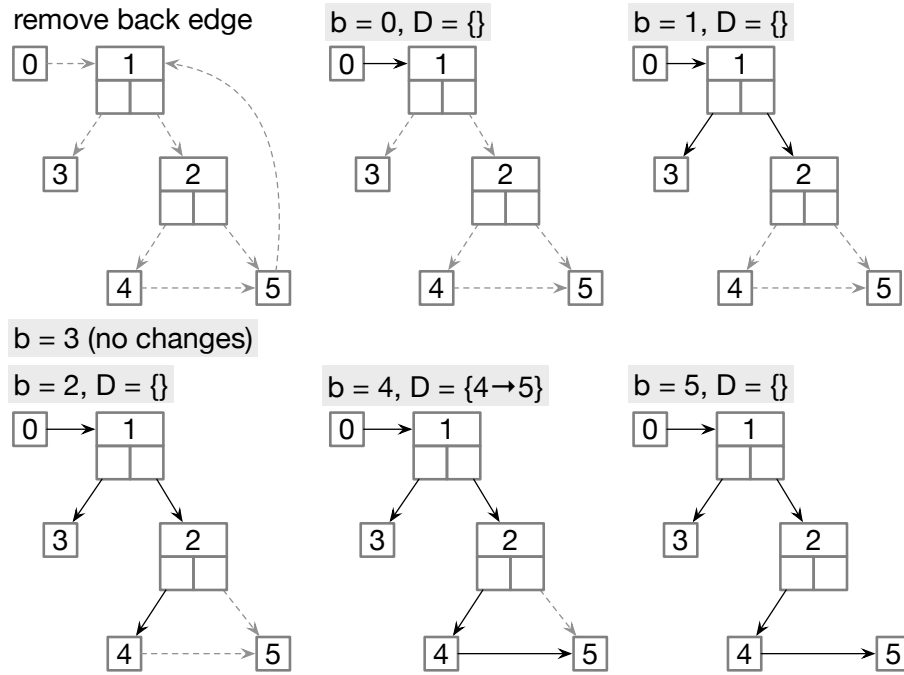


Figure 5.3. Sequence of steps that function `linearize` in Figure 5.2 performs on the program from Figure 5.1, given that `Index = [0, 1, 3, 2, 4, 5]`.

5.1.1 Properties of PCFL

In the remaining of this chapter, we will refer to two key properties of Moll and Hack [2018]’s partial control-flow linearization. Therefore, for the sake of completeness, we shall restate such properties. For more details about the proofs, we refer the reader to Moll and Hack’s original presentation. The first property that we revisit talks about the correspondence between paths in the original CFG G and in the partially linearized CFG G_l , and it is stated as follows:

Theorem 5.1 (Path Embedding [Moll and Hack, 2018, Theorem 3.2]). *For each path $\pi \in G$, there is a path $\pi_l \in G_l$ such that π is a subpath of π_l . By subpath, we mean that the nodes seen in π can be found in π_l in the same order that they appear in π .*

The second property from Moll and Hack [2018] that we revisit is related to the post-dominance relation in G_l . We write $u \succeq^{PD} v$ for post dominance in the original graph G and $u \succeq_l^{PD} v$ for post dominance in the linearized graph G_l .

Lemma 5.1 (Post Dominance for Deferral Edges [Moll and Hack, 2018, Lemma B.3]). *Let $b \in \text{Index}$ be the block currently being visited by the loop at lines 19–32 of the PCFL*

algorithm in Figure 5.2 and let s be a block such that $s \in T$ at line 20 — in other words, s is the target of a deferral edge. Then, it follows that $s \succeq_l^{PD} b$.

5.2 Taint Analysis

Partial control-flow linearization [Moll and Hack, 2018] was first devised to eliminate *divergent* branches from programs. In the context of software security, however, we are interested in *tainted* branches. Our toy language (Figure 4.1) defines two instructions, **secret** and **public**, which we shall use to separate tainted from non-tainted variables. Definition 5.2 categorizes these concepts.

Definition 5.2 (Tainted Information). The *backward slice* of a variable x is the transitive closure of its control and data dependencies (see Definition 3.3). In this work, we consider control dependencies only when dealing with phi functions and loads (see Figure 4.1). The reasoning behind this decision is that these are the only kind of assignments that might vary depending on a secret. For phi functions, this is trivial to see, since there are multiple incoming values. Loads, in the transformed program, might vary due to the guards that we add to them to ensure memory safety. For more details, see §5.3 and §5.4.3. A variable is *tainted* if its backward slice contains a secret value. A branch whose condition uses a tainted predicate is said to be *tainted*. A basic block that ends with a tainted branch is a *tainted block*. And, finally, a loop that contains an exiting tainted block is a *tainted loop*.

There are standard static analyses that can be used to identify tainted branches [Almeida et al., 2016; Rodrigues et al., 2016]. However, these techniques are orthogonal to the transformation presented in this thesis, and shall not be discussed further. In practice, we use the information analysis of Rodrigues et al. to label variables as either tainted or non-tainted. Initially, users must indicate which inputs are secret. The next example illustrates the notion of tainted information.

Example 5.4. Input `pw` is defined as secret in Figure 4.2. Since the definition of predicate `p1` relies — indirectly, through `pw.i` — on `pw`, the conditional branch at the end of the basic block `body` is tainted. If a conditional branch is tainted, then the program contains a time leak, following an earlier definition due to Rodrigues et al..

5.3 Predication

Back in §4.2, we defined a static analysis to compute the conditions that control the execution of basic blocks. However, this static analysis only works with programs that do not contain loops. In this section, we augment that analysis to handle general programs. One of the core properties that the transformation described in this thesis delivers is related to the termination of loops. We state this property as follows:

Property 5.1 (Loop Termination). If P_l is the partial linearization of a program P , then any loop of P_l can only terminate due to non-tainted (i.e. public) predicates. If, during the execution of the original program P , a loop L would have exited through a tainted edge $u \rightarrow v$, then the rest of the iterations of L in the partially linearized program P_l shall produce no observable effects.

Due to Property 5.1, if L exits through a tainted edge $u \rightarrow v$, then, in P_l , once L terminates, node v should be the only exit node whose block condition is true. However, the block condition of node u in the remaining iterations becomes false. Consequently, the edge condition of $u \rightarrow v$, if we follow Figure 4.3, would also be false; thus, the basic block v is not guaranteed to be active (see Definition 5.3). Therefore, Figure 4.3 does not work for tainted exiting edges. In the rest of this section, we explain how to extend those definitions for blocks and edges within loops.

5.3.1 Accounting for Time

Figure 4.3 does not account for the fact that basic blocks within loops may execute multiple times. In other words, in general programs, blocks and edges may be associated with multiple conditions. We introduce this notion of *time* into edge and block conditions by associating them with a number i : $BC_i(\ell)$ corresponds to the block condition of ℓ at its i -th execution (similarly for EC_i). In this sense, the definitions seen in Figure 4.3 correspond to EC_i and BC_i , $i \geq 1$.

Tainted Exiting Edges. To solve the problem with tainted exiting edges, it suffices to visualize them as n *virtual* edges, one for each iteration that the loop will perform in P_l . If the edge condition of the i -th virtual edge is true, then this means that, in P , the loop would have exited in the i -th iteration (through that tainted edge). Thus, the

edge condition of a tainted exiting edge can be defined as the disjunction of the n edge conditions associated with those n virtual edges.

Loop Headers. Figure 4.3 must be adjusted to work with loop headers. As Example 5.5 shows, block conditions must distinguish between forward edges and back edges (see Definition 3.2).

Example 5.5. Consider the block `body` in Figure 4.2. The edge condition of `begin` \rightarrow `body` is true. Hence, if we use Figure 4.3 to compute the i -th block condition of `body`, it will always be true. As a result, `body` will always be able to produce observable effects, even when not supposed to.

Figure 5.4 shows the definition of EC_i and BC_i for, respectively, tainted exiting edges and loop headers. In the case of headers, for the first iteration, we consider only the forward edges, whereas for the remaining iterations we consider only the back edges. Example 5.6 highlights the block and edge conditions for the loop seen in Figure 4.2.

$$\frac{\text{type}(\ell \rightarrow \ell') = \text{Exiting Edge} \quad \text{tainted}(p) \quad \text{terminator}(\ell) = \text{br } p, \ell', _}{EC_{i=1}(\ell \rightarrow \ell') = BC(\ell) \wedge p}$$

$$\frac{\text{type}(\ell \rightarrow \ell') = \text{Exiting Edge} \quad \text{tainted}(p) \quad \text{terminator}(\ell) = \text{br } p, \ell', _}{EC_{i>1}(\ell \rightarrow \ell') = EC_{i-1}(\ell \rightarrow \ell') \vee (BC(\ell) \wedge p)}$$

$$\frac{\text{type}(\ell \rightarrow \ell') = \text{Exiting Edge} \quad \text{tainted}(p) \quad \text{terminator}(\ell) = \text{br } p, _, \ell'}{EC_{i=1}(\ell \rightarrow \ell') = BC(\ell) \wedge \bar{p}}$$

$$\frac{\text{type}(\ell \rightarrow \ell') = \text{Exiting Edge} \quad \text{tainted}(p) \quad \text{terminator}(\ell) = \text{br } p, _, \ell'}{EC_{i>1}(\ell \rightarrow \ell') = EC_{i-1}(\ell \rightarrow \ell') \vee (BC(\ell) \wedge \bar{p})}$$

$$\frac{\text{type}(\ell) = \text{Header} \quad \text{predecessors}_f(\ell) = \{\ell_1, \dots, \ell_n\}}{BC_{i=1}(\ell) = \bigvee_{j=1}^n EC(\ell_j \rightarrow \ell)}$$

$$\frac{\text{type}(\ell) = \text{Header} \quad \text{predecessors}_b(\ell) = \{\ell_1, \dots, \ell_n\}}{BC_{i>1}(\ell) = \bigvee_{j=1}^n EC(\ell_j \rightarrow \ell)}$$

Figure 5.4. Recursive definitions of block and edge conditions for headers and tainted exiting edges. Function predecessors_f is related to forward edges and predecessors_b to back edges. When written without subscripts, $BC(\ell)$ (similarly for EC) refers to the last execution of block ℓ .

Example 5.6. For the loop seen in Figure 4.2 (b), we have $BC_1(\text{header}) = EC_1(\text{begin} \rightarrow \text{header}) = \text{true}$. That is because there are no conditions in the path from `begin` to `header`, which means that, at the first iteration — where, for the loop header, according to Figure 5.4, we consider only forward edges — block `header` will execute. For the rest of the loop, we have $EC_1(\text{header} \rightarrow \text{body}) = p0$, $EC_1(\text{body} \rightarrow \text{latch}) = \overline{p1}$, and $EC_1(\text{body} \rightarrow \text{ret.false}) = p1$. Now, suppose that $n > 1$ and, at the first iteration, we have $p1 = \text{true}$. Then, we have $BC_1(\text{latch}) = \overline{p1} = \text{false}$. At the second iteration, we consider for the loop header only the back edge; hence, $BC_2(\text{header}) = EC_1(\text{latch} \rightarrow \text{header}) = BC_1(\text{latch}) = \text{false}$. In short, from the second iteration onwards, the block condition of `header`, `body` and `latch` will be `false`, reflecting Property 5.1. After the n iterations, we have $EC_n(\text{body} \rightarrow \text{ret.false}) = \text{true}$, since $p1$ was true at the first iteration, and, consequently, $EC_1(\text{body} \rightarrow \text{ret.false}) = \text{true}$. Hence, block `ret.false` will be executed, which coincides with the fact that, in the original program, the loop would have exited through the edge `body` \rightarrow `ret.false`.

5.3.2 Active Paths

We rely on the static analysis introduced in §4.2 and expanded in this section to define the notion of *active* blocks and *active edges*. Active blocks (similarly for active edges) are those that would be executed in the original program when given the same inputs, and thus must produce the same effects as in the original program. Definition 5.3 formalizes these two concepts. We will refer to them in the proofs of theorems.

Definition 5.3 (Active Block/Edge). Let u and v be basic blocks in a program P . Block v is said to be *active* if its block condition is true. An instruction that belongs to v is *active* whenever block v itself is active. Similarly, an edge $u \rightarrow v$ in P is said to be *active* if its edge condition is true. Finally, we say that an incoming value of a phi function is active whenever its corresponding incoming edge is active.

5.4 Rewriting System

The instructions of the partially linearized program must be rewritten, so that original and transformed programs carry out the same set of observable effects. These

rewriting rules entail a number of properties concerning the elimination of time-based side channels, which Theorems 5.2 (see §5.4.1) and 5.3 (see §5.4.3) summarize. Additionally, these transformations preserve semantics, as Theorem 5.4 states (see §5.5).

5.4.1 Control Flow

Partial Control-Flow Linearization demands a compact ordering (see Definition 5.1), which implies *dominance compactness* and *loop compactness*. The former is guaranteed by function `compact_order` seen in Figure 5.2. To attain the latter, we collapse all loops into single nodes, producing a new graph structure with two types of nodes — basic blocks and loops. Loop nodes, by construction, are compact. Hence, we can apply `compact_order` to the root CFG as well as to every loop node and then join the compact orderings obtained, as Example 5.7 shows.

Example 5.7. Figure 5.5 shows the linearization of the CFG from Figure 4.2. We first collapse the loop formed by nodes `header`, `body` and `latch` into a single node L. Loop L is tainted because one of its exiting blocks — `body` — is tainted. We then produce a compact ordering of the CFG with its loop collapsed, which Figure 5.5 (b) shows, as well as a compact ordering of L, shown in Figure 5.5 (c). Since there is no tainted branch in the loop L, the loop is left unchanged. The compact ordering for the CFG can be seen as the merge of the two compact orderings from (b) and (c): `begin, header, body, latch, ret.true, ret.false, end`.

To deal with divergent loops (i.e. loops with divergent exiting blocks), Moll and Hack [2018] merge every loop exit into a single exiting block at the end of the loop, which becomes the new (unique) loop latch. The transformed loop then terminates only when all threads do. This approach, however, leads to dummy execution of instructions that could have been avoided had the public exits been preserved. In this work, we followed a different path: we redirect every *public* exiting edge of a tainted loop to the first exit block that appears in the compact ordering of the basic blocks. The following example further clarifies our partial linearization:

Example 5.8. Figure 5.5 (d) shows the linearization of the CFG with the loop collapsed. Notice that there is now a single edge leaving the loop L. This edge corresponds to every public exiting edge of a loop; in this example, there is only one from `header` to `ret.true`, but there could be more. Fig. 5.5 (e) shows the final version of the CFG.

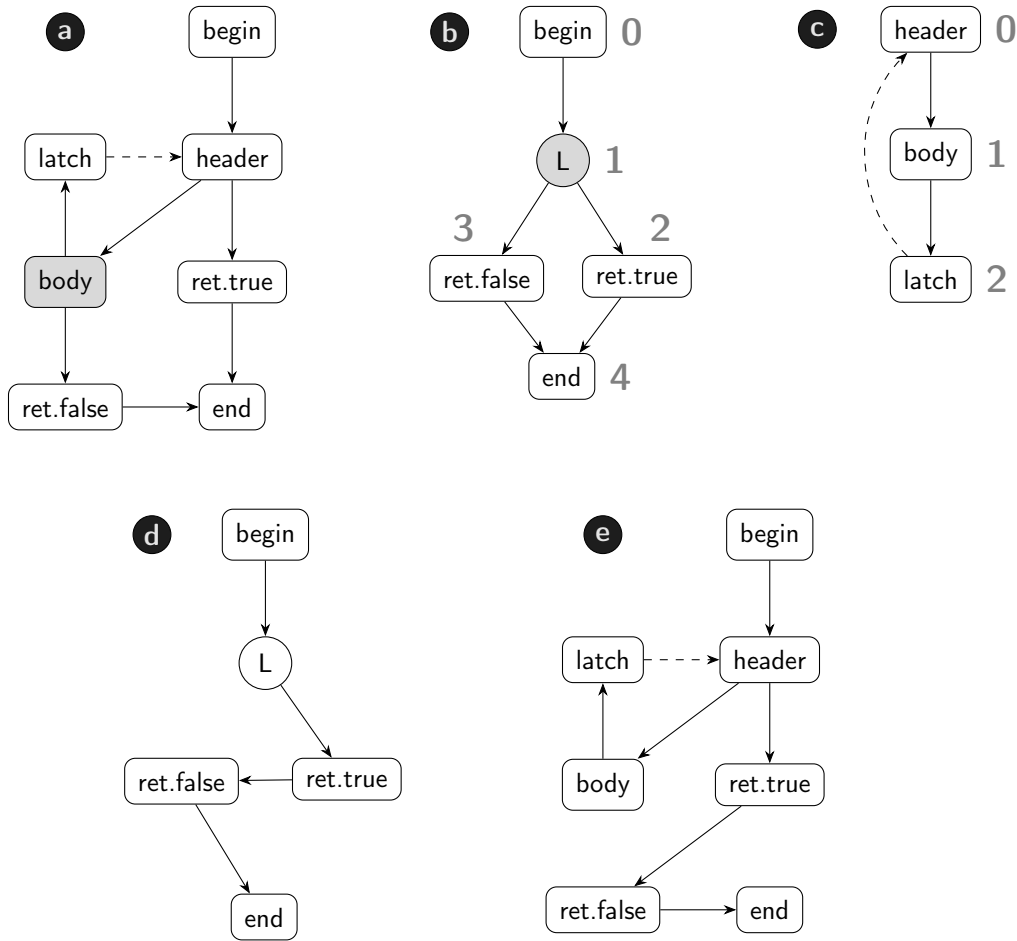


Figure 5.5. (a) CFG from Figure 4.2; dashed arrows represent back edges; gray nodes are tainted. (b) The CFG with collapsed loop; numbers indicate the compact ordering. (c) Compact ordering of loop. (d) The collapsed CFG after linearization. (e) Whole CFG after linearization.

In Example 5.8, the tainted branch at node `body` does not exist after linearization. Thus, the transformed CFG is operation invariant according to Definition 5.4: regardless of the secret inputs, it runs the same sequence of instructions. This observation is not exclusive to Example 5.8. As Theorem 5.2 states, partial linearization ensures operation invariance with regard to secret inputs. The proof of Theorem 5.2 relies on two auxiliary properties stated by Lemmas 5.2 and 5.3.

Definition 5.4 (Operation Invariance). Let $\mathbb{I} = (\mathbb{S}, \mathbb{P})$ be the inputs taken by a program P , where \mathbb{S} and \mathbb{P} are powersets of, respectively, the set of secret and public inputs. Let $\mathcal{P} \in \mathbb{P}$ be an arbitrary instance of the public inputs and $C^{\mathcal{P}} : \mathbb{S} \rightarrow \mathcal{T}$ be a deterministic channel mapping instances of the secret inputs to traces of operations observed from the execution of P . P is said to be operation invariant if $C^{\mathcal{P}}(\mathcal{S}_1) = C^{\mathcal{P}}(\mathcal{S}_2)$, for every $\mathcal{S}_1, \mathcal{S}_2 \in \mathbb{S}$. In other words, $C^{\mathcal{P}}$ is channel 1, i.e. the channel that leaks nothing.

Lemma 5.2 (Post Dominance for Tainted Branches). *Let u be a tainted block and let v_1*

and v_2 be the successors of the basic block u in the original program P . Then, after PCFL either $v_1 \succeq_l^{PD} v_2$ or $v_2 \succeq_l^{PD} v_1$.

Proof. Notice that either edge $v_1 \rightarrow v_2$ or edge $v_2 \rightarrow v_1$ become a deferral edge at line 31 of Figure 5.2, depending on which of them comes first in the compact ordering. Since the proof is the same for both cases, we will assume the first scenario: v_1 comes first in the compact ordering and thus $v_1 \rightarrow v_2$ becomes a deferral edge. When node v_1 is visited, we have $(v_1, v_2) \in D$ and thus $v_2 \in T$ at line 20 of Figure 5.2. Hence, from Lemma 5.1 it follows that $v_2 \succeq_l^{PD} v_1$. \square

Lemma 5.3 (PCFL Induces a Single Trace of Operations). *Let $\mathbb{I} = (\mathbb{S}, \mathbb{P})$ be the inputs taken by a program P , where \mathbb{S} and \mathbb{P} are powersets of, respectively, the set of secret and public inputs, and let $\mathcal{P} \in \mathbb{P}$ be an arbitrary instance of the public inputs. Furthermore, let u and v be blocks in P such that $v \succeq^{PD} u$ and assume that u is tainted. Then, given \mathcal{P} , after PCFL there is a single trace τ of operations from u to v for every $\mathcal{S} \in \mathbb{S}$.*

Proof. The proof will be by induction on the number of operation traces from block u to block v in the original program P :

Base case: If there is no tainted branch in the influence region of node u (see Definition 3.6), then, for a fixed instance \mathcal{P} of \mathbb{P} , there are exactly two subtraces τ_1 and τ_2 of operations, each one starting with one of the successors w_1 and w_2 of u . Assuming that w_1 comes first than w_2 in the compact ordering (the proof is the same for the opposite scenario), we know that the edge $u \rightarrow w_2$ will be removed and, from Lemma 5.2, we know that $w_2 \succeq_l^{PD} w_1$. Hence, the instructions from w_2 to v will be merged into τ_1 — the trace in P that starts with w_1 — forming a single trace τ in P_l .

Induction step: Let W be the set of tainted blocks in the influence region of node u that are not further nested, i.e. blocks that have nesting level equals to the nesting level of u plus one. By induction, there is exactly one trace τ_w in P_l from every $w \in W$ to their post-dominators. By combining these *disjoint* traces, following the paths in P_l , we get two traces τ_1 and τ_2 starting with each one of the successors of u . Then, the same reasoning that we used for the base case applies and we get a single trace τ from u to v in P_l . \square

Corollary 5.1 (Local Operation Invariance). *Let $\mathbb{I} = (\mathbb{S}, \mathbb{P})$ be the inputs taken by a program P , where \mathbb{S} and \mathbb{P} are powersets of, respectively, the set of secret and public inputs. Let u and v be blocks in P such that $v \succeq^{PD} u$ and assume that u is tainted. Then, after PCFL the region from block u to block v in P_l is operation invariant.*

Theorem 5.2 (PCFL Gives Operation Invariance). *Let P_l be the partial linearization of a program P . Then, program P_l is operation invariant.*

Proof. Let V be the set of tainted blocks with nesting level equals zero in the original program P (i.e. the outermost branches in P). From Lemma 5.3, we know that, for a fixed instance \mathcal{P} of \mathbb{P} (the set of public inputs), there is exactly one trace τ_v of operations in P_l for every $v \in V$. Hence, by joining these disjoint traces, we get a single trace τ for the entire program P_l . Therefore, for a fixed instance \mathcal{P} , the sequence of instructions in P_l is the same regardless of $\mathcal{S} \in \mathbb{S}$, meaning that $C^{\mathcal{P}}$ maps every instance $\mathcal{S} \in \mathbb{S}$ to the same trace $\tau \in \mathcal{T}$ with probability 1. \square

Corollary 5.2 (Trace Relations). *Let P_l be the partial linearization of a program P . Let τ_1 and τ_2 be traces of memory addresses that correspond to the execution of P when given instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$, $\mathcal{S}_1 \neq \mathcal{S}_2$. Since P_l is operation invariant, it follows that $|\tau_1| = |\tau_2|$ and, for any i , the accesses to the addresses $\tau_1[i]$ and $\tau_2[i]$ are caused by the same instruction.*

5.4.2 Phi Functions

As we have explained in §4.1, our transformation has been designed to operate on programs in the Static Single Assignment (SSA) Form. The SSA representation uses phi functions to join definitions of variables names that, in the original — pre-SSA transformation — code, would represent the same memory location. The partial linearization of a program P may lead to invalid phi functions in P_l , for a predecessor of a block v in P may not be a predecessor of v in P_l . Example 5.9 illustrates this issue.

Example 5.9. Figure 5.6 (a) shows a CFG before linearization, with block \mathbf{b} marked as tainted. Fig. 5.6 (b) shows that CFG after PCFL. Linearization removes the edge $d \rightarrow g$, hence reducing the arity of the phi function at g from three to two arguments. The rest of this section will explain how the phi function must be rewritten.

The rules that rewrite phi nodes rely on three helper functions *split*, *fill* and *fold*, which Figure 5.7 defines. Function *split*, in Figure 5.7 separates the arguments of a phi node into two sets K and R . K contains the arguments of the phi function whose corresponding blocks still are predecessors of ℓ — the block that contains the phi node — in P_l . Let φ be the phi function to be transformed and φ_l the new phi function that shall replace φ in P_l . Arguments in K are safe to be transported to φ_l without any

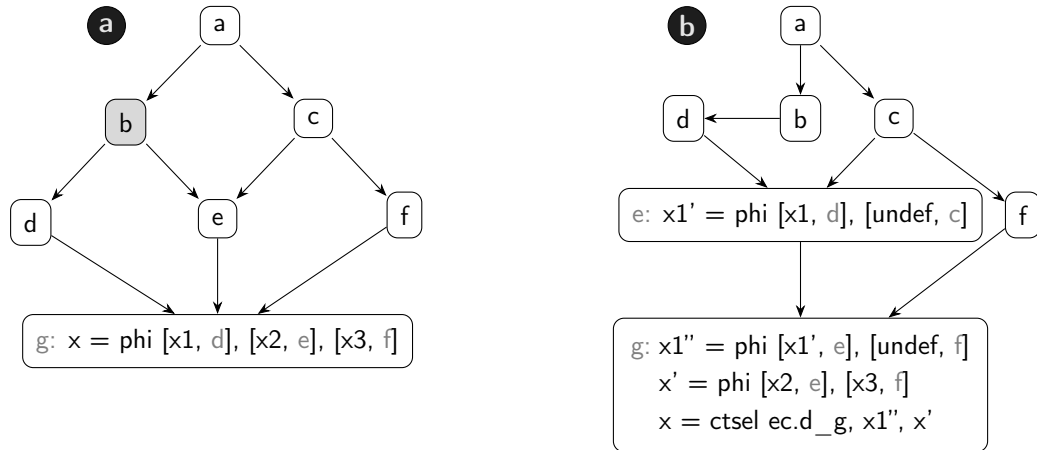


Figure 5.6. (a) CFG before partial linearization; block **b** is tainted. (b) CFG after partial linearization, with the phi function at block **g** rewritten; variable $ec.d_g$ stores the edge condition of edge $d \rightarrow g$ from the original graph (see Definition 4.1, Figures 4.3 and 5.4).

modifications, since the predecessor relation has not changed; thus, we fill as much as possible of the arguments of φ_l with K .

Function *fill* is responsible for moving the old arguments from φ to φ_l (unstarred version, third rule). Once all pairs from K were consumed, we start filling the arguments of φ_l with set R . However, the basic blocks in R are not predecessors of ℓ in P_l anymore. Hence, we need to adjust the pairs from R before attaching them as arguments of φ_l . We proceed as follows: if there is a block ℓ_j in R that reaches ℓ' (a predecessor of ℓ in G_l) in the graph G (unstarred version, first rule) then we replace ℓ_j — the original predecessor associated with value x_j — with ℓ' . However, x_j might not be always available in ℓ' , which could potentially break the SSA constraints. Hence, we must guarantee that x_j is defined when the program flows to ℓ' ; we encapsulated that as function *SSAfy*. If there is no such a block ℓ_j (unstarred version, second rule), we associate the (new) predecessor ℓ' with a special value **undef**, meaning that there is no incoming value for φ that relates to block ℓ' . The starred version of *fill* applies *fill* to all the predecessors of ℓ in G_l and returns the arguments from φ that have not yet been linked to φ_l .

The third and final step for rewriting a phi function is to link with φ_l those arguments from φ that are still unlinked. This is accomplished with function *fold*, which uses the block conditions (see Definition 4.1) of the old predecessors of ℓ to conditionally select between values. The transformation of phi functions is thus the composition of *split*, *fill* and *fold*, and it is represented by function *rewrite $_\phi$* in Figure 5.7. Function *rewrite $_\phi$* returns the variable that shall hold the value of φ in P_l ; it is worth noting, however, that there are intermediate instructions that must be added to the basic block as well. Example 5.10 illustrates the transformation of phi nodes.

Example 5.10. The phi function at block **g** in Figure 5.6 (a) is rewritten in Figure 5.6

$$\begin{array}{c}
\frac{K = \{[e_i, \ell_i] \mid \ell_i \rightarrow \ell \in E(G_l)\} \quad R = \{[e_i, \ell_i] \mid \ell_i \rightarrow \ell \notin E(G_l)\}}{\text{split}(\mathbf{x} = \mathbf{phi}[e_1, \ell_1], \dots, [e_n, \ell_n] @ \ell, G_l) = (K, R)} \\
\\
\frac{\nexists [e_i, \ell_i] \in K : \ell_i \in (\ell' \cup \ell_s) \quad \exists [e_j, \ell_j] \in R : \text{reach}_G(\ell_j, \ell')}{\text{fill}(\ell' \cup \ell_s, K, R, G) = [\text{SSAfy}(x_j), \ell']} \\
\\
\frac{\nexists [e_i, \ell_i] \in K : \ell_i \in (\ell' \cup \ell_s) \quad \nexists [e_j, \ell_j] \in R : \text{reach}_G(\ell_j, \ell')}{\text{fill}(\ell' \cup \ell_s, K, R, G) = [\mathbf{undef}, \ell']} \\
\\
\frac{\exists [e_i, \ell_i] \in K : \ell_i \in \ell_s}{\text{fill}(\ell_s, K, R, G) = ([e_i, \ell_i], K, R)} \quad \frac{}{\text{fill}(\emptyset, K, R, G) \stackrel{*}{=} (\emptyset, K \cup R)} \\
\\
\frac{\text{fill}(\ell_s, K, R, G) = ([e_i, \ell_i], K', R') \quad \text{fill}(\ell_s \setminus \ell_i, K', R', G) \stackrel{*}{=} (A, U)}{\text{fill}(\ell_s \neq \emptyset, K, R, G) \stackrel{*}{=} ([e_i, \ell_i] \cup A, U)} \\
\\
\frac{\mathbf{y}_i = \mathbf{ctsel} EC(\ell_i \rightarrow \ell), \text{SSAfy}(e_i), x}{\text{fold}(x @ \ell, [e_i, \ell_i] \cup U) \stackrel{*}{=} \text{fold}(\mathbf{y}_i @ \ell, U)} \quad \frac{}{\text{fold}(x @ \ell, \emptyset) \stackrel{*}{=} x} \\
\\
\frac{\text{split}(\mathbf{x} = \mathbf{phi} \dots @ \ell, G_l) = (K, R) \quad \text{fill}(\text{predecessors}_{G_l}(\ell), K, R, G) \stackrel{*}{=} (\{[e_1, \ell_1], \dots, [e_k, \ell_k]\}, U) \quad \text{fold}(\mathbf{x} = \mathbf{phi}[e_1, \ell_1], \dots, [e_k, \ell_k] @ \ell, U) \stackrel{*}{=} \mathbf{z}}{\text{rewrite}_\phi(\mathbf{x} = \mathbf{phi} \dots @ \ell, G, G_l) = \mathbf{z}}
\end{array}$$

Figure 5.7. Transformation rule for phi nodes. $\text{inst}@l$ indicates that the instruction belongs to the block labeled by l . $E(G)$ are the edges of graph G . fold relies on edge conditions (see Definition 4.1, Figures 4.3 and 5.4).

(b). It is almost intact, except for the argument $[\mathbf{x1}, \mathbf{d}]$, because the edge $d \rightarrow g$ was deleted. A ctsel instruction links the erased argument with the new phi function. The ctsel is parameterized by the edge condition of $d \rightarrow g$, which in Figure 5.6 (b) is encoded as variable ec.d_g . Two new phi nodes are created to preserve the SSA property, since $\mathbf{x1}$ may not be available in blocks \mathbf{c} and \mathbf{f} .

5.4.3 Memory Operations

Previously, in §4, we were linearizing the entire control-flow graph. Consequently, we needed to rewrite all memory operations lying in the influence region of *any* conditional branch. In this chapter, however, we are only partially linearizing CFGs. Therefore, only

memory operations control dependent on secret need to be modified. As in §4.3.3, we will rely on the *influence region* (Definition 3.6) of basic blocks to identify which instructions require interventions. We thus apply the rules from Figure 4.9 to loads and stores in the influence region of *tainted* blocks (Definition 5.2). Example 5.11 depicts the influence region of a tainted block. Examples 5.12 and 5.13 wrap up the transformation of loads and stores on partially linearized programs.

Example 5.11. The influence region of the tainted block `body`, from Figure 4.2, is the set formed by blocks in paths from `body` to its post-dominator `end`: `body`, `latch`, `header`, `ret.true` and `ret.false`. `body` is within its own influence region because it is inside a loop and thus there are paths from `body` to `end` that go through `body` itself.

Example 5.12. The load `pw.i = pw[i0]` in Figure 4.2 must be transformed, because it lays within the influence region of a tainted block (see Example 5.11). Let `bc.body` store the block condition of `body` and `pw.size` store the size of `pw`. Then, following function *rewrite_{ld}* from Figure 4.9, we have the transformation that takes the original code in Figure 5.8 (a) to produce the rewritten code in Figure 5.8 (b).

<p>a <code>pw.i = pw[i0]</code></p>	<p>c <code>pw[i0] ← x</code></p>
<p>b <code>c = i0 < pw.size</code> <code>c' = bc.body c</code> <code>j0 = ctsel c', i0, 0</code> <code>a = ctsel c', pw, shadow</code> <code>pw.i = a[j0]</code></p>	<p>d following load seen in part (b): <code>x' = ctsel bc.body, x, pw.i</code> <code>a[j0] ← x'</code></p>

Figure 5.8. (a) Original load from Figure 4.2. (b) Transformed load. (c) Example of a store. (d) Transformed store.

Example 5.13. Suppose that we had a store like `pw[i0] ← x` in block `body` in Figure 4.2. Following function *rewrite_{st}* from Figure 4.9, we first create a safe load, as shown in Example 5.12. For convenience, let us reuse `pw.i`. The store will then be rewritten from the original code seen in Fig. 5.8 (c) into the sequence in Fig. 5.8 (d).

Like previous work [Borrello et al., 2021; Cauligi et al., 2019], we cannot transform a program that contains memory indexation that depends on secret inputs. In other words, we cannot transform the following code: `int foo(secret v, int m[]):`

`return m[v]`. Quoting Cauligi et al. [2019], the above code is not “*publicly safe*”. Definition 5.5 augments this concept with the notion of *shadow safety*. Shadow safety is not part of Cauligi et al. work, because FaCT programs are publicly safe by construction, in contrast to C. Definition 5.6, then, formalizes the notion of data invariance — the data-related counterpart of operation invariance (Definition 5.4) — which, as stated by Theorem 5.3, is guaranteed whenever the original program is publicly safe. In the proof of Theorem 5.3 and henceforth, we write $\llbracket e \rrbracket$ to indicate the evaluation of e .

Definition 5.5 (Safety). A program meets *shadow safety* if, for every memory access $m[i]$, the *index* i is neither data nor control dependent on **secret**. If, in addition, for every access $m[i]$ that is control dependent on **secret**, $i < \mathbf{size}(m)$ can be statically proven, then this program meets *public safety* [Cauligi et al., 2019].

Example 5.14. Function `oFdF` in Figure 1.1 (a) is shadow safe: variable `i`, used to index memory accesses at line 6, is not dependent on any secret. However, the accesses at line 6 are control dependent on the predicate $\mathbf{g}[i] \neq \mathbf{pw}[i]$, which relies on the secret input `pw`. Thus, this program will be publicly safe if it can be proven that $i < \mathbf{size}(\mathbf{g}) \wedge i < \mathbf{size}(\mathbf{pw})$. The proof technique is immaterial to this discussion.

Definition 5.6 (Data Invariance). Let $\mathbb{I} = (\mathbb{S}, \mathbb{P})$ be the inputs taken by a program P , where \mathbb{S} and \mathbb{P} are powersets of, respectively, the set of secret and public inputs. Let $\mathcal{P} \in \mathbb{P}$ be an arbitrary instance of the public inputs and $C^{\mathcal{P}} : \mathbb{S} \rightarrow \mathcal{T}$ be a deterministic channel mapping instances of the secret inputs to traces of memory addresses observed from the execution of P . P is said to be data invariant if $C^{\mathcal{P}}(\mathcal{S}_1) = C^{\mathcal{P}}(\mathcal{S}_2)$, for every $\mathcal{S}_1, \mathcal{S}_2 \in \mathbb{S}$. In other words, $C^{\mathcal{P}}$ is channel 1, i.e. the channel that leaks nothing.

Theorem 5.3 (The Data Contract). *Let $T(P) = P'$ be the partial linearization of a program P with instructions rewritten as described in §§5.4.2 and 5.4.3. If program P is publicly safe, then P' is data invariant. If program P is shadow safe, then either P' is data invariant or there exist two input instances $\mathcal{I}_1 = (\mathcal{S}_1, \mathcal{P})$ and $\mathcal{I}_2 = (\mathcal{S}_2, \mathcal{P})$, $\mathcal{S}_1 \neq \mathcal{S}_2$, with corresponding traces of memory addresses τ_1 and τ_2 such that $\tau_1[i] = \mathbf{shadow}$ or (exclusive) $\tau_2[i] = \mathbf{shadow}$, for some i .*

Proof. Let \mathcal{P} be an instance of the public inputs taken by the partially linearized program P' . Then, for every pair of instances \mathcal{S}_1 and \mathcal{S}_2 of the secret inputs, we have that $C^{\mathcal{P}}$ maps \mathcal{S}_1 and \mathcal{S}_2 to, respectively, τ_1 and τ_2 , with probability 1. Recall that, from Corollary 5.2, we have $|\tau_1| = |\tau_2|$ and, for any i , $\tau_1[i]$ and $\tau_2[i]$ must correspond to the same instruction, although their values — i.e. addresses — might not be equal. Let $\tau_1[i] = a$, $\tau_2[i] = a'$ and let $\mathbf{m}[j]$ be the combination of the base address and the index that caused such memory accesses. We shall split the proof into two cases:

P' is publicly safe: We know that $j < \text{size}(\mathbf{m})$. Hence, even if the access to $\mathbf{m}[j]$ is not active, the original address will always be selected in the **ctsels** that define \mathbf{m} and j (*rewrite_{st}*, Figure 4.9). Therefore, $a = a'$ and, given that our choice of the memory access was arbitrary, $\tau_1 = \tau_2$. Thus, the theorem holds — i.e. P' is data invariant, since C^P maps \mathcal{S}_1 and \mathcal{S}_2 to the same traces.

P' is shadow safe: Let \mathcal{S}_1 and \mathcal{S}_2 be instances of the secret inputs such that $a \neq a'$ — otherwise, the theorem already holds. By inspecting rule *rewrite_{st}* (Figure 4.9) we know that the only possible values for $\llbracket \mathbf{m}[j] \rrbracket$ are the original address from P or **shadow**. Since we assumed $a \neq a'$, either address a or a' — but not both — must be the shadow memory.

□

From Theorem 5.3, the only way to have **shadow** as one of the addresses accessed by P' is to have indices larger than the known size of the associated buffer.

5.4.4 Final Example

Figure 5.9 shows the transformed code that we obtain after applying the techniques discussed in this chapter onto the code seen in Figure 4.2. Variables `g.size` and `pw.size` hold the sizes of the arrays `g` and `pw` whenever the function is invoked (Figure 5.9 (a)). If the length of the array is not known statically, then its `size` variable is initialized with zero. Notice that, as mentioned in §4.3.4, the length does not have to be a constant: it can be a symbolic expression. Whenever an expression used to index array `array` lies, in the original program, within the influence region of a tainted branch, we compare such an expression against the length of that array. This is the case of loads `g[i]` and `pw[i]` in the basic block `body`. If the comparison returns false and the block condition is false, the special variable `shadow` is used as a surrogate address (Figure 5.9 (d)). Figure 5.9 (b) shows the computation of the block condition of `header`, while (c) shows the computation of the edge condition of `body` \rightarrow `ret.false`, following the description from §5.3. Finally, Figure 5.9 (e) shows the transformation of the phi node seen in block `end` of Figure 4.2, as described in §5.4.2.

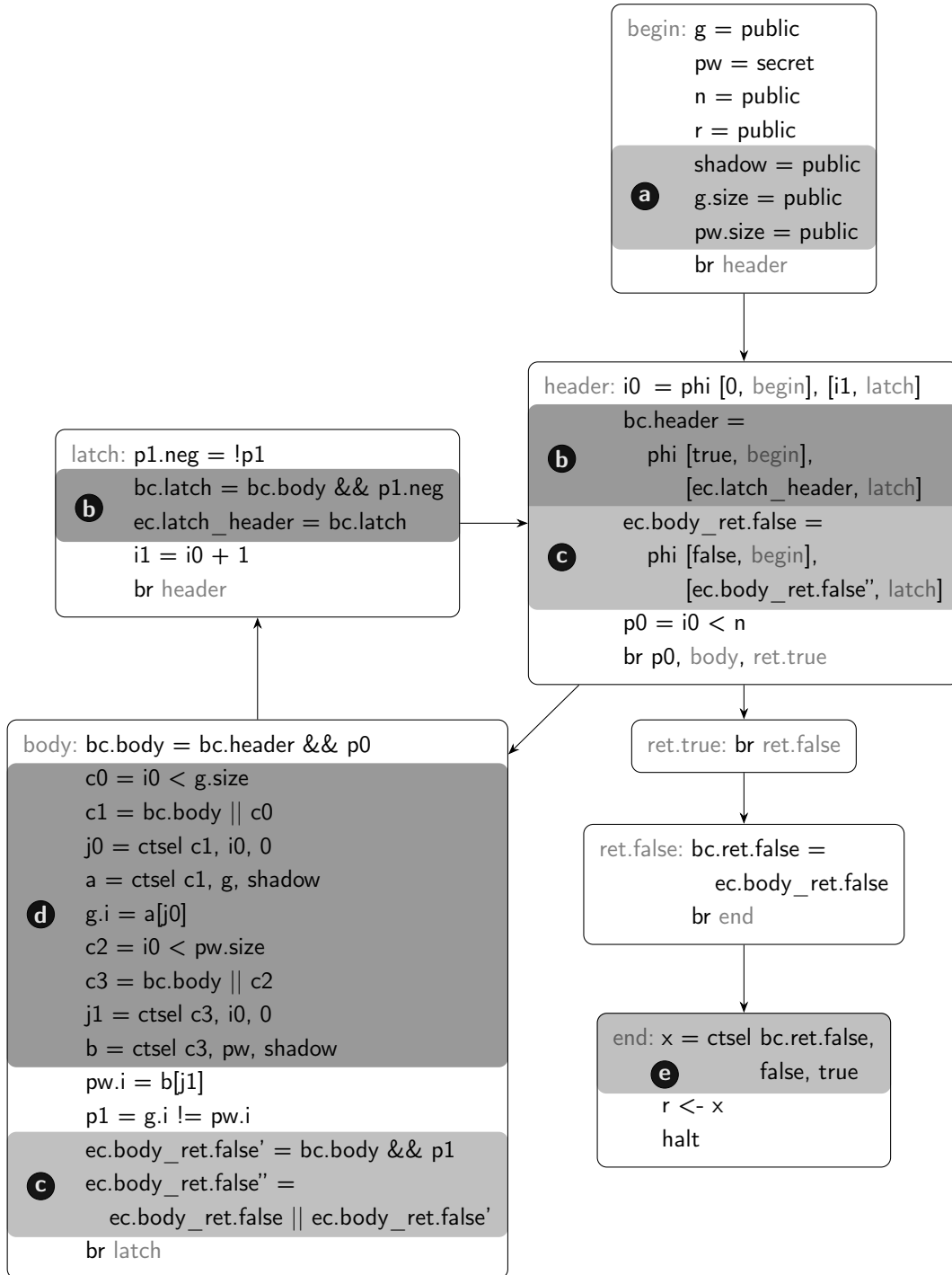


Figure 5.9. CFG from Figure 4.2 after PCFL and with the instructions rewritten.

(a) The shadow memory and the size of inputs g and pw , used in the transformation of tainted loads. (b) Computation of the block condition of the loop header (§5.3, Figure 5.4). (c) Computation of the edge condition of the tainted exiting edge $body \rightarrow ret.false$ (§5.3, Figure 5.4). (d) Predication of load instructions to ensure memory safety (§5.4.3, Figure 4.9). (e) Transformation of a phi function (§5.4.2, Figure 5.7).

5.5 Correctness

In this section, we will demonstrate that our code transformation is correct. That is, the program that we generate produces the same set of observable effects as the original program, when given the same inputs. This property is stated by Theorem 5.4. We also claim that, for shadow-safe programs, our transformation never produces a repaired code worse (i.e. that leaks more) than the original version (Theorem 5.5). This is essentially the concept of refinement from QIF, early introduced in §3.2. Furthermore, if the original program is publicly safe, then its transformed version will be both operation and data invariance, which implies isochronicity, as stated by Theorem 5.6.

5.5.1 Isochronification Preserves Semantics

The proof of correctness will be constructed on top of a few auxiliary Lemmas. Lemma 5.4 says that PCFL, as described in this thesis, does not add to nor remove any blocks from a CFG. We write $V(G)$ for the vertices of a graph G .

Lemma 5.4 (PCFL Preserves Basic Blocks). *Let G_l be the partial linearization of CFG G . Then, $V(G) = V(G_l)$.*

Proof. The proof follows directly from the PCFL algorithm defined in Figure 5.2 and the approach for handling tainted loops described in §5.4.1. \square

Lemma 5.5 states that any block executed in program P has a corresponding block that is active (Definition 5.3) in the linearized program P_l when given the same input. Recall that we write $\langle \dots \rangle$ for ordered sequences.

Lemma 5.5 (Active Trace). *Let P_l be the partial linearization of P and let τ be the trace of blocks executed in P when given an arbitrary input. Then, there exists a unique trace of blocks τ_l executed in P_l when given the same input such that, for every block $v \in \tau$, it follows that $v \in \tau_l$ and v is active in P_l .*

Proof. The proof will be by induction on τ :

Base case: In the base case, there exists a single block $v \in \tau$, which consequently is the first block of P . Since the first block has no predecessors in P and we do not add to nor remove any blocks from P_l — as stated by Lemma 5.4 —

the block condition of v is true. Therefore, block v will be active in P_l (see Definition 5.3).

Induction step: Let $\tau = \langle v_1, \dots, v_{k-1}, v_k \rangle$. By induction, we know that there exists τ_l executed in P_l such that v_i is active, $1 \leq i < k$. Let $v_j \in \tau$, $j < k$, be the predecessor of v_k that was executed in P . Since the edge $v_j \rightarrow v_k$ was taken, we know that the edge condition of edge $v_j \rightarrow v_k$ is true, and from the way block conditions are computed (Figures 4.3 and 5.4) — i.e. based on edge conditions — it follows that the block condition of v_k is true as well. Hence, we have that v_k is active in P_l . Furthermore, from Theorem 5.1, we know that v_k will be reachable from v_j in P_l . Thus, since we are considering the same input for P and P_l , it must be that $v_k \in \tau_l$, which is the unique trace for such an input. □

The next lemma is about the equivalence between the expressions in the original program P and the transformed program P' . It will be used in the proof for Theorem 5.4 (correctness). We write $\llbracket e \rrbracket$ to indicate the evaluation of e . When writing $:=$ in assignments, $:=$ can be either $=$ (for general assignments) or \leftarrow (for stores).

Lemma 5.6 (Expression Equivalence). *Let $T(P) = P'$ be the partial linearization of program P with instructions rewritten as described in §§5.4.2 and 5.4.3. Let $x := e$ be an assignment in P and let $x' := e'$ be the counterpart assignment of x in P' . Then, for any instance of the inputs such that $x := e$ is executed in P , it follows that $x' := e'$ is active in P' and $\llbracket e \rrbracket = \llbracket e' \rrbracket$.*

Proof. Consider an arbitrary instance of the inputs and let τ be the ordered sequence of assignments in P when given such an input. From Lemma 5.5, any block executed in P is active in P' . Hence, if $x := e \in \tau$, its counterpart $x' := e'$ will be active in P' , since they belong to the same block. It remains to show the correspondence between expressions e and e' . The proof will be by structural induction on the multiple assignment forms:

- (a) **x = public | secret:** These special assignments only indicate that x corresponds to an input and the transformation described in this chapter never really touches them. Hence, if the assignment is active in P' , the value assigned will be the same as in P , for we are considering the same inputs for both the original program P and the repaired version P' .
- (b) **x = e:** Let expression e be composed by subcomponents v_1, \dots, v_n . Then, by induction on each v_i , $1 \leq i \leq n$, we know that $\llbracket v_i \rrbracket = \llbracket v'_i \rrbracket$, where v'_i is the counterpart of v_i in P' . Thus, it must be that e evaluates to the same value in both P and P' , when given the same inputs.

- (c) $\mathbf{x} = \mathbf{m}[i]$: From rule $rewrite_{ld}$ seen in Figure 4.9, we know that this assignment will be rewritten as $\mathbf{x} = \mathbf{a}[i']$. But, since the assignment is active, i.e. the block condition is true, it follows that $\mathbf{a} \equiv \mathbf{m}$ and $i' \equiv i$. That is, the assignment that will be performed is the same in both P and P' . By induction we know that the values of \mathbf{m} and i are the same in P and P' , when given the same input. Hence, the memory access in P' , under the described circumstances, will be the same as in P . It remains to show that the value stored in that address is the same in P and P' , which follows by induction.
- (d) $\mathbf{m}[i] \leftarrow e$: From rule $rewrite_{st}$ (Figure 4.9), we know that this assignment will be rewritten as $\mathbf{a}[i'] \leftarrow e'$. Since the assignment is active in P' , we have $e' \equiv e$, $\mathbf{a} \equiv \mathbf{m}$ and $i' \equiv i$. By the same reasoning used in case (b), we can conclude that $\llbracket e \rrbracket = \llbracket e' \rrbracket$. Thus, since the memory region accessed in P and P' will be the same, the value stored in this address, after the store operation is performed, will be the same in P and P' .
- (e) $\mathbf{x} = \mathbf{phi} [e_1, \ell_1], \dots, [e_n, \ell_n]$: First notice that there cannot be two edge conditions $EC(\ell_i \rightarrow \ell)$ and $EC(\ell_j \rightarrow \ell)$, $i \neq j$, that are true at the same time, since two edges cannot be taken simultaneously. From function $rewrite_{\phi}$ defined in Figure 5.7, we know that the edge conditions are used as the conditions of **ctsel** instructions that link incoming values that did not fit in the transformed phi function. The counterpart of \mathbf{x} in P' will either be the transformed phi function or the last **ctsel** created, if any. Let $\ell_i \rightarrow \ell$ be the active edge (i.e. edge condition is true). Then, either the incoming value e_i still is an incoming value in the transformed phi function, and consequently the condition of every **ctsel** will be false, or there will be a single **ctsel** whose condition is true and which will select e_i . In both cases, we have $\llbracket \mathbf{phi} [e_1, \ell_1], \dots, [e_n, \ell_n] \rrbracket = \llbracket e' \rrbracket$.
- (f) $\mathbf{x} = \mathbf{ctsel} c, v_t, v_f$: This kind of assignment is never modified by the transformation described in this chapter. Thus, it suffices to show that c , v_t and v_f evaluate to the same values in P and P' , which follows directly by induction.

□

With Lemmas 5.5 and 5.6, we can prove the correctness of our transformation:

Theorem 5.4 (Semantics). *Let $T(P) = P'$ be the partial linearization of program P with instructions rewritten as described in §§5.4.2 and 5.4.3. If P' terminates, then programs P and P' produce the same set of observable effects.*

Proof. The only instruction in our toy language that can produce observable effects is a store. From function $rewrite_{st}$ defined in Fig. 4.9, we have the following three cases:

Store is active: Let the store be of the form $a' \leftarrow e'$ and let $a \leftarrow e$ be its counterpart in P . Since $a' \leftarrow e'$ is active in P' , the original memory region is accessed — i.e. $a \equiv a'$ — and the store updates the state of the memory. By Lemma 5.6, we have $\llbracket e \rrbracket = \llbracket e' \rrbracket$. Therefore, the state of a in both P and P' is updated with the same value, thus causing the same effects.

Store is not active \wedge access is safe: The original memory region is accessed, but the value to be assigned is replaced with the current value stored in that memory address; hence, the store is silent and no effect can be observed — i.e. the state of the memory does not change after the store is executed.

Store is not active \wedge access is not safe: The original store is replaced with a store to **shadow** and the value to be assigned is replaced with the value currently stored in **shadow**; hence, the store is performed silently and no effect can be observed.

It remains to show that every store executed in program P is active in program P' , which follows from Lemma 5.5. \square

5.5.2 Isochronification Implements Refinement

We now move our attention to the property of refinement. Theorem 5.5 states that, whenever the original program P is shadow safe, the code that results from the transformation described in this thesis never leaks more than P .

Theorem 5.5 (Refinement). *Let $T(P) = P'$ be the partial linearization of a program P with instructions rewritten as described in §§5.4.2 and 5.4.3, and assume P is shadow safe. Let $\mathbb{I} = (\mathbb{S}, \mathbb{P})$ be the inputs taken by P , where \mathbb{S} and \mathbb{P} are powersets of, respectively, the set of secret and public inputs. Finally, let $\mathcal{P} \in \mathbb{P}$ be an arbitrary instance of the public inputs. Then, if $C^{\mathcal{P}}: \mathbb{S} \rightarrow \mathcal{T}$ and $D^{\mathcal{P}}: \mathbb{S} \rightarrow \mathcal{T}'$ are deterministic channels mapping the secret inputs to traces of operations and memory addresses observed from the execution of, respectively, P and P' , it follows that $C^{\mathcal{P}} \sqsubseteq_{\circ} D^{\mathcal{P}}$.*

Proof. By Theorem 5.2, P' is operation invariant. If P is publicly safe, then, by Theorem 5.3, P' is data invariant. In this case, $D^{\mathcal{P}}$ maps every instance $\mathcal{S} \in \mathbb{S}$ to a single trace $\tau \in \mathcal{T}'$, inducing a partition with a single cell, which is coarser than that induced by $C^{\mathcal{P}}$; thus, $C^{\mathcal{P}} \sqsubseteq_{\circ} D^{\mathcal{P}}$. If P is not publicly safe, then, by hypothesis, it is at least shadow safe. Since P' is operation invariant, for a fixed instance $\mathcal{P} \in \mathbb{P}$, the instructions

executed are all the same, regardless of the secret inputs. Thus, we shall focus on the memory addresses only. Let $\mathcal{S} \in \mathbb{S}$ be an arbitrary instance of the secret inputs, where $\mathcal{S} = \{s_1, \dots, s_n\}$. Pick a memory access $m[j]$ control dependent on s_i , $1 \leq i \leq n$, such that there exists one possible value of s_i that triggers an access to the shadow memory in P' — if such a memory access does not exist, P' is data invariant and, as observed before, the theorem holds. Then, in the original program P , there are exactly two options, according to the value of s_i : either the address $m[j]$ is in the trace or it is not. In P' , on the other hand, the memory access will always be executed. However, there is again exactly two scenarios: either the original address or the shadow memory will be accessed. Therefore, the partitions induced by $C^{\mathcal{P}}$ and $D^{\mathcal{P}}$ will be the same, and thus we can conclude that $C^{\mathcal{P}} \sqsubseteq_{\circ} D^{\mathcal{P}}$. \square

If the original program is not only shadow safe, but also publicly safe (Definition 5.5), then the repaired code meets isochronicity; that is, regardless of the sensitive inputs, the transformed program runs the same sequence of instructions and accesses the same sequence of memory addresses. This property is formally described as follows:

Theorem 5.6 (Isochronicity). *Let $T(P) = P'$ be the partial linearization of a publicly-safe program P with instructions rewritten as described in §§5.4.2 and 5.4.3. Then, P' is both operation and data invariant.*

Proof. Follows directly from Theorems 5.2 and 5.3. \square

Chapter 6

Evaluation

This chapter evaluates the techniques described in this thesis through five research questions:

- RQ1:** By how much does the proposed approach increase code size?
- RQ2:** What is the running time of applying the proposed transformation onto programs?
- RQ3:** How does the proposed approach impact the running time of programs?
- RQ4:** What are the security guarantees achieved by the proposed approach?
- RQ5:** How the general C programs compiled with the proposed approach compare with programs written in a domain-specific language for constant-time cryptography?

We report numbers for both our loop-free method (presented in §4), which we call as “Lif”, and our general technique (described in §5), which we refer to as “PCFL”. To provide perspective on our results, we compare them with those produced by **FaCT** [Cauligi et al., 2019], **SC-Eliminator** [Wu et al., 2018] and **Constantine** [Borrello et al., 2021]. The last two tools aim at making programs data and operation invariant with regard to secret inputs. Our implementations, in turn, only guarantee operation invariance, although when handling publicly safe programs they also ensure data invariance. Code written in the **FaCT** domain-specific language is, by construction, publicly safe; hence, **FaCT** delivers both operation and data invariance for this class of programs. When presenting results for **SC-Eliminator** and **Constantine**, we show how these tools fare with and without data-flow protection.

Hardware. Experiments run on an Intel(R) Core(TM) i5-1035G1 4-Core processor, clocked at 3.6 GHz. L1 data and instruction caches have 128 KB. Main memory has 8 GB.

Software. The above hardware runs in Linux Manjaro 21.2.5 (5.16.14-1-generic x86_64). Our program transformations are implemented in LLVM 13.0. We use a version of `Constantine` downloaded from <https://github.com/pietroborrello/constantine>, which is implemented in LLVM 9.0. `SC-Eliminator` is available as an ACM artifact at <https://zenodo.org/record/1299357>, using LLVM 3.9.1. However, due to problems to reuse that artifact, we have downloaded `SC-Eliminator` directly from <https://bitbucket.org/mengwu/timingsyn>, and have updated it to use LLVM 13.0. We use a version of `FaCT` downloaded from <https://github.com/PLSysSec/FaCT>, which uses libraries from LLVM 6.0. These tools were downloaded on December 6th, 2021. Therefore, results from this paper can be directly compared with `SC-Eliminator`, using LLVM 13.0 to obtain the original bytecode that will be transformed. `Constantine`, on the other hand, uses LLVM 9.0; hence, the starting bytecode file is not guaranteed to be the same.

To check if the transformed benchmarks run in constant time, we used `CTgrind`, a `Valgrind` [Nethercote and Seward, 2007] plugin available at <https://github.com/agl/ctgrind>. In addition to linearizing the control-flow graph of programs, `Constantine` and `SC-Eliminator` also try to eliminate memory-based side channels. `SC-Eliminator` does it by preloading data in the beginning of functions; `Constantine` does it by traversing the entire buffer whenever a cell within said buffer is read or written. Such interventions make code much slower and much bigger. Thus, for fairness, we show results for these tools with and without data preloading. It is our understanding that, once data linearization is disabled, `SC-Eliminator` and `Constantine` work for the same purpose as our implementation of PCFL.

Benchmarks. We use 13 benchmarks, including seven from Wu et al. [2018]. Each benchmark has at least two inputs, with parts tagged as either public or secret. Two of the benchmarks, `ss13` and `donna`, are from the `FaCT` repository. We have translated them into C. We implemented the four remaining benchmarks: `hash-one`, `plan-many`, `plain-one` and `log-redactor`, to exercise control-flow constructs absent in Wu et al.’s collection. The benchmark `plain-one` corresponds to the example that we have been using throughout the paper (Figure 4.2). Programs `hash-one` and `plain-many` are variations of `plain-one`. As a side note, we handle, without any user intervention, six of the seven programs used in Borrello et al.’s and Wu et al.’s evaluation that contain any control flow branching on sensitive data. The only exception is `loki91`, because its original implementation contains a loop whose every exit is controlled by sensitive information; hence, it is inherently leaky. Numbers reported in this chapter refer to the transformed program, optimized with LLVM `opt -O3`. The evaluation of security guarantees (§6.5) uses these optimized programs, in binary format. The benchmarks contain code to initialize inputs and print outputs; however, our numbers refer only to their kernels. `Lif` and `SC-Eliminator`

cannot transform three benchmarks due to unbounded loops: `donna`, `ssl3` and `loki91`. Benchmark `plain-many`, when transformed by `Constantine`, `Lif` and `SC-Eliminator`, crashes at running time. Furthermore, for benchmark `ssl3`, `Constantine` produces code whose output is incorrect.

6.1 RQ1: Size of Transformed Code

Figure 6.1 reports the size of the programs produced by the different tools that we evaluate in this paper. Size is measured by counting LLVM instructions in the intermediate representation of the transformed kernel after optimizations run.

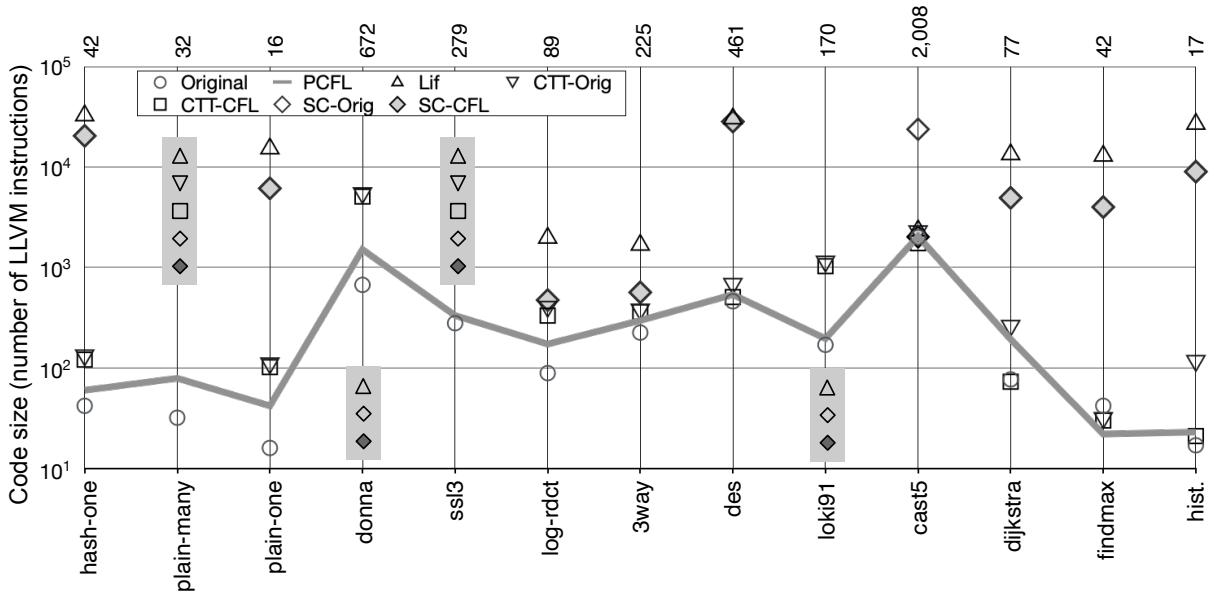


Figure 6.1. Code size (in number of LLVM instructions) of transformed programs. Numbers on top show the size of original programs (compiled with LLVM 13.0 at the `-O3` optimization level). Symbols in gray boxes show tools that are missing for particular benchmarks. `CTT` refers to `Constantine`, `SC` refers to `SC-Eliminator`. `Orig` refers to these two tools as originally implemented. `CFL` refers to these two tools with control-flow linearization only — thus, closer to our implementation of `PCFL` in purpose.

Considering only the nine benchmarks that `SC-Eliminator` and `Lif` handle, our implementation of `PCFL` generates 3,393 LLVM instructions. The original version of `SC-Eliminator` produces 97,772 instructions, whereas `SC-Eliminator`’s `CFL` gives 76,004. `Lif` generates 138,079 instructions. It is worth remembering that both `SC-Eliminator` and `Lif` were applied to a version of the programs with loops entirely

unrolled, for neither of the two tools can handle general loops; hence the higher number of instructions. The complete implementation of `Constantine` (CFL + DFL) yields 4,182 LLVM instructions, while `Constantine`'s CFL outputs 3,293. When compiled with LLVM 13.0 at `-O3`, the original 13 benchmarks add up to 4,130 instructions, 2,977 of which corresponds to the subset of nine benchmarks handled by all tools. In relative terms, our partial linearization increases code size by 1.33x. Considering only the 11 benchmarks correctly handled by `Constantine`, `Constantine`'s original and CFL-only implementations increase code size by, respectively, 2.73x and 2.64x.

6.2 RQ2: Transformation Time

Figure 6.2 shows the time (in milliseconds) that each technique evaluated in this paper takes to transform programs. To provide the reader a baseline, we also show the time that LLVM takes to apply all the optimizations at the `-O3` level onto these programs. Numbers refer only to the time taken by `opt`, LLVM's optimizer, to run passes onto LLVM intermediate representation: it does not include time to parse C or to generate machine code — roughly the same for all the approaches.

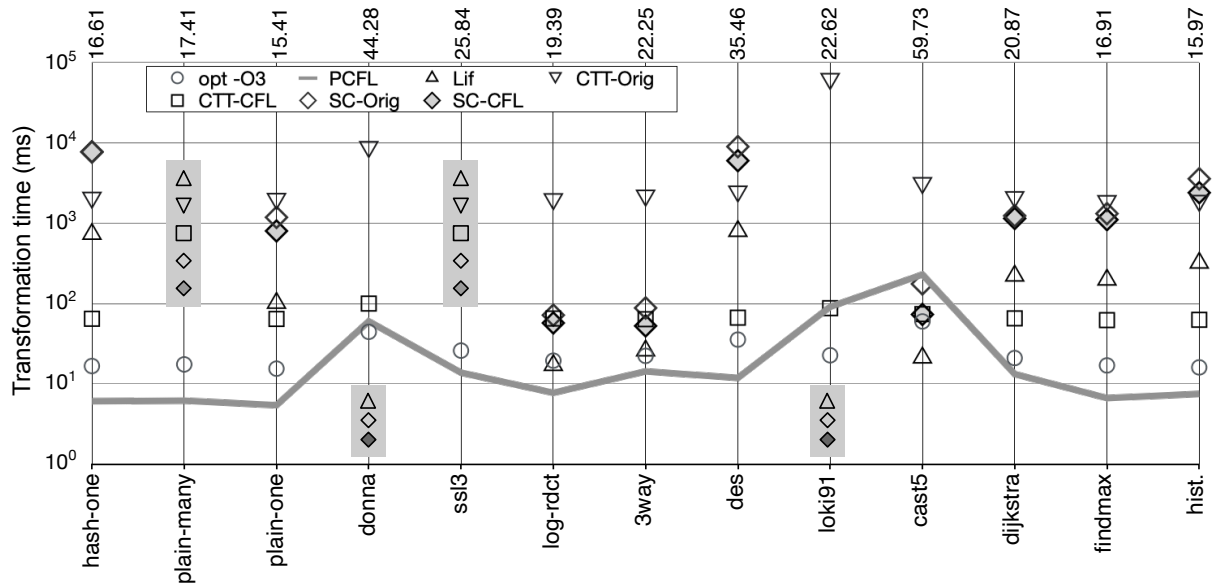


Figure 6.2. Time (in milliseconds) to apply each transformation onto the benchmarks. To give the reader some perspective on this comparison, the numbers on top show the time taken to run LLVM `opt -O3` on each benchmark. The gray boxes mark benchmarks that some tools could not handle.

Considering only the nine benchmarks that all tools can deal with, it takes, on

average, 24.73 ms to apply `opt -O3` onto each benchmark, without any transformation. Our PCFL technique takes about 33.49 ms per benchmark. `Lif`, not counting the time to unroll loops, takes 271.61 ms. The original implementation of `SC-Eliminator` takes 2,697.06 ms, whereas `SC-Eliminator`'s CFL takes 2,149.28 ms. `SC-Eliminator` and `Lif` are slower because they operate on larger programs, due to unrolling. `Constantine`'s CFL takes about 65.19 ms per benchmark. However, when we consider the entire script used to apply `Constantine` — which includes everything from profiling up to all the transformations that it applies — this number grows up to 2,045.22 ms. This total is the summation of several independent passes that `Constantine` applies — some of them coming from LLVM's accompanying tools. Thus, LLVM bytetimes are read and traversed multiple times. We understand that were these passes grouped into a single LLVM pass, `Constantine` could run faster.

6.3 RQ3: Performance of Transformed Code

Figure 6.3 shows the running time of transformed programs. Timing the benchmarks was challenging: except for `loki91`, they all run under less than 1 ms. We executed each benchmark 20 times per tool, removed the two fastest and the two slowest samples per benchmark and averaged the remaining 16 samples. We then used Student's T-Test [Gosset, 1908] to check for statistically significant results across all the three populations, pair-wisely. Assuming a confidence level of 99%, and considering only the nine benchmarks that `SC-Eliminator` and `Lif` can handle, we could observe five results where the programs produced by our implementation of PCFL run faster than those generated by `SC-Eliminator`'s CFL and six results where our approach performs better than both `SC-Eliminator`'s original implementation and `Lif`, our loop-free method. Lowering the confidence interval to 0.95, we observe one additional benchmark for which our PCFL technique yields code that runs faster than the code produced by `SC-Eliminator`'s CFL. The mean overhead introduced by both `SC-Eliminator` and our PCFL is of 1.61x. The CFL-only version of `SC-Eliminator` adds less overhead onto programs: 1.42x. `Lif`, on the other hand, generates code that is 3.74x slower. Nonetheless, the expressive overhead promoted by `Lif` was hugely influenced by the benchmark `des` (8.32x slower).

Assuming a confidence level of 99%, our PCFL method performs better than both `Constantine`'s original and CFL-only implementations in six out of the 11 benchmarks correctly handled by `Constantine`. By lowering the confidence interval to 0.95, we could observe one additional benchmark for which we produce faster code than `Constantine` (both versions). The overhead introduced by `Constantine` with respect to the nine bench-

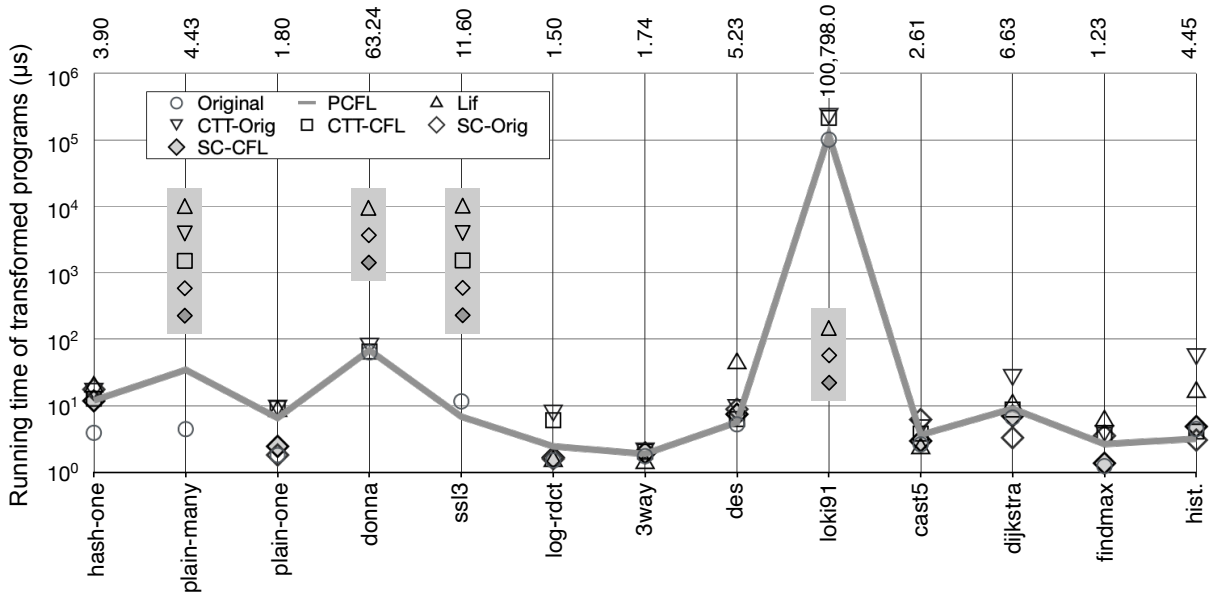


Figure 6.3. Running time (in microseconds) of transformed programs. Numbers on top show time of original programs (compiled with LLVM 13.0). Symbols in gray boxes show missing tools for particular benchmarks.

marks that all tools can correctly deal with was larger than ours: 4.62x for the original implementation and 1.94x for Constantine’s CFL-only version. Similarly to the case of Lif, the huge overhead introduced by Constantine’s original tool was heavily influenced by the benchmark `histogram` (12.26x slower). If we consider the 11 benchmarks that Constantine handles, then its average overhead is of 2.24x (original) and 2.12x (CFL), whereas the overhead imposed by our implementation of PCFL is of 1.17x — which is also the mean overhead caused by our tool if we consider all the 13 benchmarks.

6.4 RQ4: Security Evaluation

Figure 6.4 summarizes the security guarantees met by code produced by the different tools considered in this work. Figure 6.4 evaluates data and operation invariance. To verify the latter, we relied on CTgrind [Langley, 2010], a Valgrind plugin that determines if a program contains a branch that reads data tainted by secret information. CTgrind reports that SC-Eliminator failed to achieve operation invariance for two benchmarks: `hash-one` and `histogram`. We analyzed the LLVM intermediate files produced by SC-Eliminator and confirmed that SC-Eliminator indeed failed to linearize some of the tainted branches. In several benchmarks, CTgrind reports that code produced by

`Constantine` is not operation invariant. Debugging the code produced by `Constantine` is harder than analyzing the code produced by `SC-Eliminator`, because `Constantine` applies a more extensive range of transformations on the target program. We inspected the code that `Constantine` produced for `plain-only`, our smallest kernel. In that case, `CTgrind`’s result seems to be a false positive: `Constantine` uses memory cells whose purpose is similar to our shadow memory. These blocks of memory are not initialized but might be used in conditional operations, albeit never modifying the observable state of the program. In this case, `CTgrind` issues warnings because it considers as tainted any memory that is not initialized.

	Original			PCFL				Lif				SC-Eliminator				Constantine			
	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3	Cor	Data	Opr	Opr3
hash-one	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N	N
plain-many	N	N	N	Y	N	Y	Y	X	X	X	X	X	X	X	X	X	X	X	X
plain-one	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	N
donna	N	N	N	Y	Y	Y	Y	UL	UL	UL	UL	UL	UL	UL	UL	Y	N	N	N
ssl3	Y	N	N	Y	Y	Y	Y	UL	UL	UL	UL	UL	UL	UL	UL	N	N	N	N
log-rdct	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	N
3way	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
des	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y
loki91	Y	N	N	Y	Y	Y	Y	UL	UL	UL	UL	UL	UL	UL	UL	Y	Y	Y	Y
cast5	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	Y	Y	Y
dijkstra	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	Y	N	N
findmax	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
histogram	N	N	N	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	Y	Y	N	N	N

N Property not verified
 Y Property verified
 UL Unbounded loop
 X Tool crashes or transformed program crashes

Figure 6.4. Security guarantees achieved by the different tools. `Cor` indicates if the transformed program produces the same output as its original counterpart (i.e. if the transformed program is *correct*). `Data` refers to data invariance. `Opr` refers to operation invariance without compiler optimizations. `Opr3` refers to operation invariance at the LLVM `opt -O3` optimization level.

When probing data variance in Figure 6.4, we use an LLVM instrumentation pass to verify if the sequence of addresses accessed by each kernel is the same, regardless of the input. In this case, we consider the original versions of `Constantine` and `SC-Eliminator`, not the versions that only do control-flow linearization. In several cases, we observe that neither tool achieves data invariance in our “string-of-addresses” threat model. This apparent failure is a consequence of the threat model that these tools assume: they consider all the memory accesses to a cache line as the same access. `SC-Eliminator` reads every buffer at the beginning of the kernel. `Constantine` replaces every memory access like `a[i]` with a loop that reads the entire array `a`. Yet, to be practical, both tools use strided accesses to scan arrays. Thus, an access such as `a[i%n]` will remain data variant after transformed by either `SC-Eliminator` or `Constantine`, as long as `n` is smaller than the size of the cache line. In both cases, the size of the cache line is set as part of the implementation of the tools, and changing it requires recompilation. Our failures to achieve complete data invariance, in turn, are due to the fact that some memory accesses

are replaced with the shadow memory, which is a unique address for the entire program. Nevertheless, we could verify that the data contract stated in Theorem 5.3 holds for all the 13 benchmarks. In particular, we ported `ssl3` and `donna` from FaCT. When written in FaCT, every benchmark is publicly safe, and we succeed in delivering the same guarantees as that domain-specific language: complete non-interference between secret inputs and addresses accessed in the instruction and data caches.

6.5 RQ5: Comparison with a Domain-Specific Language

Our implementation of partial-control flow linearization in LLVM in practice gives developers the chance to obtain in C (or other languages that LLVM supports) the same security guarantees provided by FaCT [Cauligi et al., 2019]. However, contrary to C, FaCT deals with less general control-flow constructs. Currently, FaCT supports three control-flow statements: `if-then-else`; `for-from-to` and `return`. Our implementation of PCFL, in contrast, handles the whole of the LLVM intermediate representation, whose unstructured control flow subsumes the entire C grammar, including constructs absent in FaCT, such as `do-while`, `break`, `continue` and `goto`¹. Only four out of the 13 benchmarks that we evaluate in this paper admit straightforward translation between C and FaCT: `plain-one`, `plain-hash`, `openssl-ssl3` and `donna`. The last two were taken from the FaCT repository and translated to C. Figure 6.5 compares the code produced for these programs.

The programs written in FaCT are, usually, shorter and faster. However, the programs that we generate contain code to ensure memory safety, like the use of the shadow memory, as in Figure 5.8 (b). Therefore, every linearized load operation in a program produced with our technique will contain four extra instructions absent in the equivalent FaCT program. Moreover, the transformation of a store operation requires a load to read the current value in memory. Because this load is also safe, the four-instruction overhead also applies. In FaCT, developers use a type qualifier, `assume(e)`, which let them specify that expression e is the upper bound of an array. This clause works as a contract: the compiler does not generate code to ensure in-bounds accesses; rather, the programmer promises that buffer limits will be respected. Thus, it is possible to provoke out-of-bounds access in the FaCT program, because contracts are not verified. To this effect, we have forced out-of-bounds accesses in `plain-one` — something that cannot happen in the bi-

¹PCFL has one restriction concerning the `goto` statement: *loops must be natural*. The implication of this fact is that PCFL will not linearize a loop with multiple entry points. In the C programming language, such loops can be created with `goto`.

	PCFL					FaCT				
	Time (μ s)	.o	Instrs.	.o	Instrs.	Time (μ s)	.o	Instrs.	.o	Instrs.
ssl3	7.99	3,352	333	4,160	369	6.97	3,288	514	4,064	609
donna	53.61	11,384	1511	12,032	1543	59.44	7,576	774	6,920	804
plain-one	12.66	1,024	36	1,696	67	5.47	704	27	1,600	70
hash-one	11.96	1,112	60	1,856	97	4.43	904	84	1,816	129
		without main		with main			without main		with main	

Figure 6.5. Comparison between programs written in C and linearized with PCFL, and similar programs written in FaCT, using equivalent control-flow structures. The column `.o` shows the size, in bytes, of the binary object file. The column `Instrs` shows the number of instructions in the LLVM representation of each program. Because they use different `main` functions, we show results with and without this routine.

nary produced with PCFL.

Chapter 7

Conclusion

During these two years, we developed a code transformation technique to eliminate time-based side channels. This transformation ensures that a program always runs the same sequence of operations, regardless of the sensitive inputs it takes. In addition, if a program satisfies certain constraints — at least shadow safety (Definition 5.5) — then the repaired code never leaks more than its original counterpart (Theorem 5.5).

A key contribution of this work is the adaptation of a technique first developed in the context of vectorization — Moll and Hack [2018]’s partial control-flow linearization — to solve an open question in software security: the *static* elimination of side channels in general programs, while preserving branches controlled only by public information. We believe that, for publicly safe programs (Definition 5.5), the techniques discussed in this thesis let a programmer write, directly in general-purpose languages like C, code that meets the same safety properties as algorithms written in the FaCT [Cauligi et al., 2019] domain-specific language.

In contrast to previous techniques [Wu et al., 2018], our approach is capable of transforming programs with unbounded loops, as long as these loops contain at least one exiting edge that depends only on public data (Property 5.1). As opposed to the work of Borrello et al. [2021], the transformation described in §5 does not require profiling information, and yet it handles as many programs as **Constantine**. Furthermore, as shown in §§6.1 and 6.3, code generated by our tools are competitive when compared to previous work [Wu et al., 2018; Borrello et al., 2021], and the transformation itself has practical compilation time (§6.2).

It is worth remembering that, contrary to Wu et al.’s and Borrello et al.’s works, in this thesis we did not present any specific method to mitigate cache-based leaks (e.g. preloading). Even though we discussed throughout the thesis the concept of data invariance and its importance in implementations of cryptographic routines, our transformation is focused on the elimination of instruction-based leaks; i.e. it ensures operation invariance (Definition 5.4). Nevertheless, for publicly safe programs, our transformation also delivers data invariance (Definition 5.6). This last guarantee is a consequence of the fact that publicly safe programs that are operation invariant are also data invariant, a property that follows from Theorem 5.6. We believe that, considering the current state-of-the-art

techniques, the most efficient approach would be to combine our static control-flow linearization scheme with Borrello et al.'s data-flow linearization technique to provide full protection against time-based leaks.

Bibliography

- Agat, J. (2000). Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, page 40–53, New York, NY, USA. Association for Computing Machinery.
- Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7):1–19. ISSN 0362-1340.
- Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. (1983). Conversion of control dependence to data dependence. In *POPL*, page 177–189, New York, NY, USA. Association for Computing Machinery.
- Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016). Verifying constant-time implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 53–70, USA. USENIX Association.
- Almeida, J. B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., and Strub, P. (2020). The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982, New York, NY, USA. IEEE.
- Alvim, M. S., Chatzikokolakis, K., McIver, A., Morgan, C., Palamidessi, C., and Smith, G. (2020). *The Science of Quantitative Information Flow*. Springer.
- Appel, A. W. (1997). *Modern Compiler Implementation in Java*. Cambridge University Press, USA. ISBN 0521583888.
- Balliu, M., Dam, M., and Guanciale, R. (2014). Automating information flow analysis of low level code. In *CCS*, page 1080–1091, New York, NY, USA. Association for Computing Machinery.
- Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., and Trieu, A. (2019). Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL).
- Barthe, G., Grégoire, B., Laporte, V., and Priya, S. (2021). Structured leakage and applications to cryptographic constant-time and cost. In *CCS*, page 462–476, New York, NY, USA. Association for Computing Machinery.

- Barthe, G., Grégoire, B., and Laporte, V. (2018). Secure compilation of side-channel countermeasures: The case of cryptographic “constant-time”. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343.
- Besson, F., Dang, A., and Jensen, T. (2019). Information-flow preservation in compiler optimisations. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 230–23012. IEEE.
- Borrello, P., D’Elia, D. C., Querzoni, L., and Giuffrida, C. (2021). Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS*, page 715–733, New York, NY, USA. Association for Computing Machinery.
- Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R. S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., and Stefan, D. (2019). Fact: A dsl for timing-sensitive computation. In *PLDI*, page 174–189, New York, NY, USA. Association for Computing Machinery.
- Chattopadhyay, S. and Roychoudhury, A. (2018). Symbolic verification of cache side-channel freedom. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2812–2823.
- Chen, Y., Mendis, C., Carbin, M., and Amarasinghe, S. (2021). Vegen: A vectorizer generator for simd and beyond. In *ASPLOS*, page 902–914, New York, NY, USA. ACM.
- Cleemput, J. V., Coppens, B., and De Sutter, B. (2012). Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4). ISSN 1544-3566.
- Clements, A. (2013). *Computer Organization and Architecture: Themes and Variations*. Cengage Learning, USA. ISBN 1111987041.
- Cock, D., Ge, Q., Murray, T., and Heiser, G. (2014). The last mile: An empirical study of timing channels on sel4. In *CCS*, page 570–581, New York, NY, USA. Association for Computing Machinery.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition. ISBN 0262033844.
- Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira Jr., W. (2011). Divergence analysis and optimizations. In *PACT*, page 320–329, USA. IEEE.
- Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *POPL*, page 25–35, New York, NY, USA. Association for Computing Machinery.

- Deng, C. and Namjoshi, K. S. (2017). Securing the SSA transform. In *International Static Analysis Symposium*, pages 88–105. Springer.
- Deng, C. and Namjoshi, K. S. (2018). Securing a compiler transformation. *Formal Methods in System Design*, 53(2):166–188.
- Dhem, J., Koeune, F., Leroux, P., Mestré, P., Quisquater, J., and Willems, J. (1998). A practical implementation of the timing attack. In Quisquater, J. and Schneier, B., editors, *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 167–182, Berlin, Heidelberg. Springer-Verlag.
- Fell, A., Pham, H. T., and Lam, S.-K. (2019). Tad: Time side-channel attack defense of obfuscated source code. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC '19*, page 58–63, New York, NY, USA. Association for Computing Machinery.
- Ferrante, J. and Mace, M. (1985). On linearizing parallel code. In *PLDI*, page 179–190, New York, NY, USA. Association for Computing Machinery.
- Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349. ISSN 0164-0925.
- Flynn, M. J. (1972). Some computer organizations and their effectiveness. *Transactions on Computers*, 21(9):948–960. ISSN 0018-9340.
- Garland, M. and Kirk, D. B. (2010). Understanding throughput-oriented architectures. *Commun. ACM*, 53(11):58–66. ISSN 0001-0782.
- Gosset, W. S. (1908). The probable error of a mean. *Biometrika*, 6(1):1–25. Originally published under the pseudonym “Student”.
- Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., and Costa, M. (2017). Strong and efficient cache side-channel protection using hardware transactional memory. In *Security Symposium*, page 217–233, USA. USENIX Association.
- Guarnieri, M., Köpf, B., Reineke, J., and Vila, P. (2020). Hardware-software contracts for secure speculation.
- Hunt, S. and Sands, D. (2006). On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90. ISSN 0362-1340.
- Karrenberg, R. and Hack, S. (2012). Improving performance of opencl on cpus. In *Compiler Construction*, page 1–20, Berlin, Heidelberg. Springer-Verlag.

- Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, page 104–113, Berlin, Heidelberg. Springer-Verlag.
- Kocher, P. C., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *CRYPTO*, page 388–397, Berlin, Heidelberg. Springer-Verlag.
- Langley, A. (2010). Checking that functions are constant time with valgrind. <https://github.com/agl/ctgrind>.
- Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, page 75, USA. IEEE Computer Society.
- Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115. ISSN 0001-0782.
- Moll, S. and Hack, S. (2018). Partial control-flow linearization. In *PLDI*, page 543–556, New York, NY, USA. Association for Computing Machinery.
- Moreira, R. E., Collange, C., and Quintão Pereira, F. M. (2017). Function call re-vectorization. In *PPoPP*, page 313–326, New York, NY, USA. Association for Computing Machinery.
- Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100. ISSN 0362-1340.
- Ngo, V. C., Dehesa-Azuara, M., Fredrikson, M., and Hoffmann, J. (2017). Verifying and synthesizing constant-resource implementations with types. In *Security and Privacy*, pages 710–728, Washington, DC, USA. IEEE.
- Rafnsson, W., Jia, L., and Bauer, L. (2017). Timing-sensitive noninterference through composition. In *POST*, page 3–25, Berlin, Heidelberg. Springer-Verlag.
- Rane, A., Lin, C., and Tiwari, M. (2015). Raccoon: Closing digital side-channels through obfuscated execution. In *SEC*, page 431–446, USA. USENIX Association.
- Reparaz, O., Balasch, J., and Verbauwhede, I. (2017). Dude, is my code constant time? In *DATE*, page 1701–1706, Leuven, BEL. European Design and Automation Association.
- Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366.
- Rodrigues, B., Quintão Pereira, F. M., and Aranha, D. F. (2016). Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 110–120, New York, NY, USA. Association for Computing Machinery.

- Sampaio, D., Souza, R. M. d., Collange, C., and Pereira, F. M. Q. a. (2014). Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4). ISSN 0164-0925.
- Singel, R. (1976). Declassified nsa document reveals the secret history of TEMPEST about (TEMPEST: A signal problem). *Cryptologic Spectrum*, 2(3):26--30.
- Soares, L., Canesche, M., and Pereira, F. M. Q. a. (2022). Side-channel elimination via partial control-flow linearization. Computer Science Department, Federal University of Minas Gerais (UFMG). Manuscript submitted for publication.
- Soares, L. and Pereira, F. M. Q. a. (2021). Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210, Washington, USA. IEEE Press.
- Tizpaz-Niari, S., Černý, P., and Trivedi, A. (2019). Quantitative mitigation of timing side channels. In Dillig, I. and Tasiran, S., editors, *Computer Aided Verification*, pages 140--160, Cham. Springer International Publishing.
- Towle, R. A. (1976). *Control and Data Dependence for Program Transformations*. PhD dissertation, University of Illinois at Urbana-Champaign, USA. AAI7624191.
- Van Cleemput, J., De Sutter, B., and De Bosschere, K. (2020). Adaptive compiler strategies for mitigating timing side channel attacks. *IEEE Transactions on Dependable and Secure Computing*, 17(1):35–49.
- Wolfe, M. J. (1978). Techniques for improving the inherent parallelism in programs – master thesis. Master’s thesis, University of Illinois at Urbana-Champaign.
- Wray, J. C. (1992). An analysis of covert timing channels. *J. Comput. Secur.*, 1(3–4):219–232. ISSN 0926-227X.
- Wu, M., Guo, S., Schaumont, P., and Wang, C. (2018). Eliminating timing side-channel leaks using program repair. In *ISSTA*, page 15–26, New York, NY, USA. Association for Computing Machinery.
- Zdancewic, S. and Myers, A. C. (2001). Robust declassification. In *CSFW*, page 5, USA. IEEE Computer Society.
- Zhang, R., Bond, M. D., and Zhang, Y. (2022). Cape: Compiler-aided program transformation for htm-based cache side-channel defense. In *Compiler Construction*, page 181–193, New York, NY, USA. Association for Computing Machinery.